TURBO PASCAL®
für Windows

WINDOWS-
PROGRAMMIER-
HANDBUCH

BORLAND

Turbo Pascal für Windows

Windows-Programmierhandbuch

BORLAND

Copyright © 1991 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Softwareprogrammes Turbo Pascal für Windows wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal für Windows liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal für Windows nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal für Windows) dürfen eigene Programme, die Sie mit dem Sprachcompiler Turbo Pascal für Windows geschrieben und compiliert haben, ohne Einschränkung oder Verpflichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1991«.

Die Beispielprogramme von Turbo Pascal für Windows, die auf den Originaldisketten mitgeliefert werden demonstrieren die verschiedenen Eigenschaften von Turbo Pascal für Windows. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz, aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Ubertragung ins Deutsche:
Marcus O. Lerch, Arne Schäpers und Borland GmbH

Typographische Konzeption des Innenteils: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

Papier: Schleipen GmbH, Bad Dürkheim. Das verwendete Papier ist säurefrei und alterungsbeständig. Die Produktion erfolgt in einem geschlossenen Wasserkreislauf ohne Einsatz von optischen Aufhellern und unter Verwendung von chlorarm gebleichten, holzfreien Zellstoffqualitäten. Die Abwasserreinigung wird in einer betriebseigenen chemisch-biologischen Kläranlage vorgenommen.

April 1991

Inhalt

Windows leicht gemacht 1

Turbo Pascal für Windows 1
Die Sprache Turbo Pascal 1
Programmieren für Windows in Windows 2
Der Ressourceneditor 2
Die ObjectWindows-Bibliothek 2
Turbo Debugger für Windows 3
Notwendige Vorkenntnisse 3
Der Inhalt dieses Handbuchs 3

I. Eine Einführung in Object-Windows 5

1. Die Vererbung des Fensters 7

Was ist eine Windows-Anwendung? 7
Vorteile von Windows 8
Anforderungen an die Hardware 8
Programmieren für Windows 9
Ereignisgesteuerte Architektur 9
Geräteunabhängige Grafik 9
Multitasking 10
Speicherverwaltung 10
Ressourcen 11
Dynamisches Linken 11
Zwischenablage 12
Dynamischer Datenaustausch 12
Multidokumentschnittstelle 12
Datentypen von Windows 13
Objektorientierte Programmierung für Windows 13
Eine bessere Schnittstelle zu Windows 13
Schnittstellenobjekte 14
Die Abstraktion von Windows-Funktionen 14
Beantworten von Botschaften 14
Die Struktur einer Windows-Anwendung 15

Die Struktur von Windows 15
Interaktion mit Windows und DOS 16
»**Hello, Windows**« 16
Der Start einer Anwendung 17
Das Hauptfenster 18
Vorgehensweise bei der Anwendungsentwicklung 18

2. Schritt für Schritt durch Windows 19

Schritt 1: Eine einfache Windows-Anwendung 20
Voraussetzungen 20
Definieren des Anwendungstyps 21
Schritt 2: Der Objekttyp für das Hauptfenster 22
Was ist ein Fensterobjekt? 23
Handles 23
Übergeordnete und untergeordnete Fenster 24
Erstellen eines neuen Fenstertyps 24
Beantworten von Botschaften 25
Beenden eines Programms 26

3. Füllen eines Fensters 29

Was ist ein Bildschirmkontext? 29
Schritt 3: Zeichnen von Text in einem Fenster 30
Botschafts-Records 31
Löschen des Bildschirms 32
Schritt 4: Zeichnen von Linien im Fenster 32
Bewegungen der Maus 32
Beantworten von Mausbewegungs-Botschaften 33
Schritt 5: Ändern der Liniendicke 35
Steuern der Liniendicke 35
Verwendung des Eingabefeldes 38
Schritt 6: Automatische Neuanzeige von Grafiken 39
Neuzeichnen von Fenstern 40
Speichern von Grafiken als Objekte 40
Neuzeichnen gespeicherter Grafiken 42

4. Hinzufügen eines Menüs 43

Menü ressourcen 43
Schritt 7: Ein Menü für das Hauptfenster 44
Abfangen der Menübotschaft 45
Beantwortung der Menübotschaft 46
Die Verknüpfung von Ressourcen zu einer ausführbaren Datei 47

5. Konstruktion eines Dialogs 49

Schritt 8: Hinzufügen eines Pop-up-Fensters 50
Erzeugen und Anzeigen des Pop-up-Fensters 50
Die Funktion MakeWindow 51
Hinzufügen eines Dialogfensters 52
Hinzufügen eines Objektfeldes 53
Ändern des Init-Konstruktors 54
Anzeigen des Dialogfensters 54
Schritt 9: Speichern der Zeichnung in einer Datei 55
Überwachung des Status 55
Speichern und Laden von Dateien 57

6. Anzeigen von Fenstern 59

Schritt 10: Anzeigen eines Hilfefensters 59
Die Verwendung von Units mit ObjectWindows 59
Ändern des Hauptprogramms 60
Erzeugen der Unit 60
Steuerelemente in einem Fenster 62
Was sind Steuerelemente? 62
Erstellen von Steuerelementen 63
Steuerelementobjekte als Felder 63
Verwalten von Steuerelementen 64
Antworten auf Steuerelement-Ereignisse 65

II. Die Verwendung von ObjectWindows 69

7. Überblick über ObjectWindows 71

ObjectWindows-Konventionen 71
Die ObjectWindows-Hierarchie 71
Basisobjekte 73
TApplication 73
Schnittstellenobjekte 73
TWindowsObject 73
Fensterobjekte 73
TWindow 73
TEditWindow 74
TFileWindow 74
Dialogobjekte 74
TDialog 74
TDlgWindow 74
TFileDialog 74
TInputDialog 74
Steuerelementobjekte 74
TControl 75

TButton 75
TListBox 75
TComboBox 75
TCheckBox 75
TRadioButton 75
TGroupBox 75
TEdit 75
TScrollBar 75
TStatic 75
MDI-Objekte 75
TMDIWindow 76
TMDIClient 76
ObjectWindows-Units 76
WObject 76
WinProcs 76
WinTypes 76
StdDlgs 76
StdWnds 77
Windows-Ressourcen 77
Windows-Funktionen 77
ObjectWindows-Aufrufe von Windows-Funktionen 77
Zugriff auf Funktionen 78
Kombinieren von Stilkonstanten 79
Typen von Windows-Funktionen 79
Schnittstellenfunktionen der Fensterverwaltung 79
Funktionen der grafischen Geräteschnittstelle (GDI) 79
Schnittstellenfunktionen der Systemdienste 79
Callback-Funktionen 79
Windows-Botschaften empfangen 80
Wie Ereignisse Methodenaufrufe werden 81
Die Parameter einer Windows-Botschaft 82
Typen von Windows-Botschaften 83
Botschaften der Fensterverwaltung 83
Initialisierungsbotschaften 83
Eingabebotschaften 83
Systembotschaften 83
Zwischenablagebotschaften 83
Systeminformationsbotschaften 83
Botschaften der Manipulation von Steuerelementen 83
Benachrichtigungsbotschaften von Steuerelementen 84
Benachrichtigungsbotschaften von Bildlaufleisten 84
Botschaften von Nicht-Client-Bereichen 84
MDI-Botschaften (Botschaften der Multidokumentschnittstelle) 84
Standardbearbeitung von Botschaften 84
Senden von Botschaften 84
Botschaftsbereiche 85
Anwenderdefinierte Botschaften 86

8. Anwendungsobjekte 87

Steuerung des Anwendungsflusses 87
Initialisieren von Anwendungen 89
Initialisieren des Hauptfensters 89
Initialisieren jeder Anwendungsinstanz 90
Initialisieren der ersten Anwendungsinstanz 91
Anwendungen ausführen 92
Anwendungen schließen 93

9. Schnittstellenobjekte 95

Der Typ TWindowsObject 95
Wofür Schnittstellenobjekte benötigt werden 96
Das Verhältnis zwischen über- und untergeordneten Fenstern 97
Listen untergeordneter Fenster 97
Iterieren über untergeordnete Fenster 98
Botschaftsverarbeitung 99
Antworten auf Botschaften 101
Befehlsabhängige Botschaften und Botschaften untergeordneter Fenster 102
Verarbeitung befehlsabhängiger Botschaften 102
Verarbeitung von Botschaften untergeordneter Fenster 103
Botschaftsverarbeitung durch Standardantworten 104

10. Fensterobjekte 107

Der Typ TWindow 107
Initialisieren und Erstellen von Fensterobjekten 108
Initialisieren des Fensterobjekts 108
Erstellen von Fensterelementen 110
Zusammenfassung 111
Fensterklassen-Registrierung 111
Registrierungsattribute 113
Das Feld für den Klassenstil 114
Das Feld für das Symbol 114
Das Feld für den Zeiger 114
Das Feld für die Hintergrundfarbe 114
Das Feld für das Standardmenü 115
Editier- und Dateifenster 115
Editierfenster 115
Dateifenster 117
Rollen von Fenstern 117
Scroller-Attribute 118
Dem Fenster einen Scroller zuweisen 119
Ein Roll-Beispiel 120
Automatisches Rollen und Permanentes Aktualisieren 121
Ändern der Rolleinheiten und des Rollbereichs 122

Ändern der Rollposition 122
Bestimmen der Seitengröße 123
Optimieren von Zeichenmethoden für das Rollen 123

11. Dialogobjekte 125

Verwendung von Dialogressourcen 125
Verwendung von Dialogobjekten 125
Konstruieren und Initialisieren von Dialogen 126
Ausführen von Dialogen 126
Modale und nicht-modale Dialoge 126
Beenden von Dialogen 127
Verwalten von Dialogobjekten 127
Manipulieren von Steuerelementen und Bearbeiten von Botschaften 128
Manipulieren von Dialog-Steuerelementen 129
Beantworten von Benachrichtigungsbotschaften der Steuerelemente 129
Beispiel für die Kommunikation zwischen Dialog- und Steuerelement 130
Zuordnen von Steuerelementobjekten 131
Aufruf von DefWndProc 131
Erweitertes Beispiel für einen Dialog 132
Dialogfenster 133
Eingabedialoge 134
Datei-Dialogfenster 135
Initialisieren von Datei-Dialogfenstern 135
Dateidialoge verwenden 136

12. Steuerelementobjekte 137

Der Objekttyp TControl 138
Konstruieren und Erstellen von Steuerelementobjekten 139
Freigeben und Entfernen von Steuerelementobjekten 140
Bearbeiten von Steuerelementen und Botschaften 140
Manipulieren des Steuerelements eines Fensters 141
Antwort auf Benachrichtigungsbotschaften der Steuerelemente 141
Fenster, die wie Dialoge agieren 143
Listen 143
Konstruieren und Erstellen von Listen 143
Ändern von Listen 144
Abfragen von Listen 145
Aus einer Liste auswählen 145
Beispielprogramm: LBoxTest 147
Statische Steuerelemente 147
Konstruieren statischer Steuerelemente 148
Informationen über statische Steuerelemente 148
Ändern von statischen Steuerelementen 149
Beispiel: Die Anwendung StatTest 149
Aktionsschalter-Steuerelemente 149
Beantworten von Schalterbotschaften 150

Auswahl- und Schaltfelder 150
Konstruieren von Auswahl- und Schaltfeldern 151
Informationen über den Status von Wahlfeldern 151
Ändern des Wahlfeld-Status 152
Gruppen 152
Beantworten von Gruppenbotschaften 153
Beispielprogramm: BtnTest 153
Bildlaufleisten 154
Konstruieren von Bildlaufleisten 154
Informationen über Bildlaufleisten 155
Ändern von Bildlaufleisten 156
Beantworten von Bildlaufleisten-Ereignissen 156
Beispiel: SBarTest 158
Editierfelder 158
Konstruieren von Editierfeldern 159
Zwischenablage und Editieroperationen 160
Implementierung 161
Informationen über Editierfelder 161
Ändern von Editierfeldern 163
Text löschen 163
Text einfügen 164
Textauswahl und Rollen 164
Beispielprogramm: EditTest 164
Kombinationsfenster 165
Drei Arten von Kombinationsfenstern 165
Einfache Kombinationsfenster 165
Aufspringende Kombinationsfenster 165
Aufspringende Listen-Kombinationsfenster 166
Auswahl einer Kombinationsfenster-Art 166
Konstruieren von Kombinationsfenstern 167
Ändern von Kombinationsfenstern 168
Beispielprogramm: CBoxTest 168
Setzen von Werten für Steuerelemente 168
Definieren eines Transferpuffers 169
Definieren des entsprechenden Dialogs oder Fensters 170
Übertragen der Informationen 171
Transfer bei frei definierten Steuerelementen 172
Transfer-Beispiel 172
Steuerelemente und die Botschaft wm_Paint 173

13. MDI-Objekte 175

Die Komponenten einer MDI-Anwendung 175
Das MDI-Fenster - ein Objekt 176
Konstruieren von MDI-Fenstern 176
Konstruieren von MDI-Rahmenfenstern 177
Konstruieren von untergeordneten MDI-Fenstern 177
Bearbeiten von Botschaften in MDI-Anwendungen 178

Verwalten von untergeordneten MDI-Fenstern 179
Aktivierung von untergeordneten Fenstern 179
Das Menü für untergeordnete Fenster 179
Beispiel für eine MDI-Anwendung 180

III. ObjectWindows für Fortgeschrittene 181

14. Speicherverwaltung 183

Die Speicherverwaltung von Windows 183
Die Verwaltung des Speichers durch Windows 184
Handles und Adressen 185
Lokaler und globaler Speicher 185
Der lokale Heap 186
Reservierung und Verwendung von lokalem Speicher 187
Freigeben und verwerfen lokaler Speicherblöcke 188
Neureservieren und Ändern lokaler Speicherblöcke 189
Abfragen lokaler Speicherblöcke 189
Hinweise für die Programmierung 190
Der globale Heap 190
Reservierung und Verwendung von globalem Speicher 190
Weitere Möglichkeiten zum Sperren eines globalen Blocks 192
Freigeben und verwerfen von globalen Speicherblöcken 192
Neureservieren und Ändern globaler Speicherblöcke 193
Abfragen globaler Speicherblöcke 193
Ändern des globalen Verwerfens 194
Speicherknappheit 194
Hinweise für die Programmierung 195

15. Dynamische Linkbibliotheken 197

Zugriff auf DLL-Routinen 197
Ein einfaches Beispiel für eine DLL 198

16. Dynamischer Datenaustausch 203

Die Zwischenablage von Windows 203
Formate für die Zwischenablage 203
Speichern von Daten in der Zwischenablage 204
Laden von Daten aus der Zwischenablage 205
Zeitverzögerte Datenkonvertierung 206
Botschaftsübermittlung zwischen Anwendungen 207
Dynamischer Datenaustausch 208
Begriffe 208
Aufbau einer Konversation 209
Beenden einer Konversation 211

Methoden für den Datenaustausch 211
Anfordern eines einzelnen Items 212
Andauernder Datentransfer 215
Den Server zum Ändern eines Datenwertes auffordern 216
Ausführen von Makrobefehlen im Server 217
Das Thema »System« 218

17. Die grafische Geräteschnittstelle 219

Bildschirmkontexte 219
Verwalten von Bildschirmkontexten 220
Inhalt des Bildschirmkontexts 220
Bitmap-Grafiken 220
Farbe 221
Abbildungsmodi 221
Clipping-Regionen 221
Zeichenwerkzeuge 222
Zeichenwerkzeuge 222
Vordefinierte Werkzeuge 222
Virtuelle Zeichenwerkzeuge 223
Virtuelle Stifte 223
Virtuelle Pinsel 224
Virtuelle Schriften 225
Grafikanzeige in Fenstern 229
Zeichnen in Fenstern 229
Verwalten eines Bildschirmkontexts 229
Aufrufen von Grafikfunktionen in Fenstern 230
Aktualisieren von Fenstern 230
Grafik-Strategie 230
Die Verwendung von Zeichenwerkzeugen 231
GDI-Zeichenfunktionen 233
Funktionen für das Zeichnen von Text 233
Funktionen für das Zeichnen von Linien 234
MoveTo und LineTo 234
Polyline 234
Arc 235
Funktionen zum Zeichen von geometrischen Formen 236
Rectangle 236
RoundRect 236
Ellipse 237
Pie und Chord 237
Polygon 238
Der Einsatz von Paletten 239
Erstellen einer Farbpalette 240
Zeichnen mit Paletten 240
Informationen über eine Palette 241
Ändern einer Palette 241
Beantworten von Palettenbotschaften 242

18. Details über Ressourcen 245

Erstellen von Ressourcen 246
Hinzufügen von Ressourcen zu einer .EXE-Datei 246
Der Ressourcen-Compiler 247
Makrodateien für den Ressourcen-Compiler 248
Hinweise für die Benutzung des Ressourcen-Compilers 251
Laden von Ressourcen in einer Anwendung 252
Laden von Menüs 252
Laden von Accelerator-Tabellen 253
Laden von Dialogfenstern 254
Laden von Mauszeigern und Symbolen 254
Laden von String-Ressourcen 255
Laden von Bitmaps 256
Erstellung eines Pinsels mit Hilfe eines Bitmaps 257
Anzeigen von Bitmaps in Menüs 259

19. Standardrichtlinien für Anwendungen 261

Entwurfs prinzipien 261
Antworten, die der Erwartung des Anwenders entsprechen 261
Anwender-Steuerung des Programmflusses 262
Standard für Erscheinungsbild und Verhalten 262
Generelle Präsentation 263
Maus- und Tastatur-Interaktionen 263
Menüs 264
Dialogfenster 265
Anmerkungen zu Entwürfen 265
Sichere Programme 266
Programmieren unter dem Motto »Alles oder Nichts« 266
Der Sicherheitsbereich 267
Weitere Fehler bei der Erstellung von Fenstern 268
Großverbraucher 269

20. Kollektionen 271

Kollektionobjekte 271
Kollektionen werden dynamisch vergrößert 271
Kollektionen sind polymorph 272
Typenprüfung und Kollektionen 272
Kollektionen mit Nicht-Objekten 272
Erstellen einer Kollektion 273
Iteratormethoden 275
Der ForEach-Iterator 275
Die Iteratoren FirstThat und LastThat 276
Sortierte Kollektionen 277
Stringkollektionen 278
Noch einmal: Iteratoren 279

Einträge suchen 280
Polymorphe Kollektionen 280
Kollektionen und Speicherverwaltung 282

21. Streams 285

Die Aufgabe: Objekt-Ein-/Ausgaben 285
Die Lösung: Streams 286
Streams sind polymorph 286
Streams bearbeiten Objekte 286
Der Einsatz von Streams 287
Streams aufbauen 287
Streams lesen und schreiben 288
Die Methode Put 288
Die Methode Get 288
Die Methode Error 289
Streams entfernen 289
Objekte für Streams 289
Die Methoden Load und Store 289
Stream-Registrierung 290
ID-Nummern von Objekten 291
Automatische Felder 291
Registrierung 292
Registrierung von Standardobjekten 292
Der Stream-Mechanismus 292
Die Put-Methode 292
Die Get-Methode 293
nil-Objektzeiger 293
Kollektionen in Streams: Ein Beispiel 293
Hinzufügen der Store-Methoden 294
Registrierungs-Records 295
Registrierung 296
Schreiben in den Stream 296
Wer speichert was? 297
Felder in Streams 297
Instanzen von nebengeordneten Fenstern 298
Streams kopieren 299
Streams mit Direktzugriff 299
Nicht-Objekte in Streams 300
Der Entwurf eigener Streams 300
Fehlerbehandlung in Streams 301

Index 303

Windows leicht gemacht

Turbo Pascal für Windows eröffnet Ihnen neue Wege für die Entwicklung von Microsoft-Windows-Anwendungen. Bisher wurde zur Programmierung fast ausschließlich die Sprache C und eine Vielzahl von Programmiertools verwendet, die unter DOS liefen. Das Resultat war ein langwieriger und komplizierter Programmierprozeß.

Turbo Pascal für Windows

Turbo Pascal für Windows bringt einige Neuerungen mit sich, die die Entwicklung von Microsoft-Windows-Programmen erleichtern:

- Erweiterungen der Sprache Pascal;
- eine Entwicklungsumgebung für den Betrieb unter Windows;
- einen ebenfalls unter Windows laufenden Ressourcen-Editor;
- die ObjectWindows-Bibliothek für die Windows-Programmierung;
- den Turbo Debugger für Windows.

Wir wollen uns diese Merkmale in den folgenden Abschnitten etwas näher ansehen.

Die Sprache Turbo Pascal

Falls Sie bereits Windows-Anwendungen programmiert haben, haben Sie hierzu wahrscheinlich die Sprache C verwendet. Es wird aber auch Situationen geben, in denen Sie nicht in C programmieren wollen (oder können), zum Beispiel wenn vorhandener Pascal-Code für eine Windows-Anwendung angepaßt werden soll. Turbo Pascal für Windows stellt Ihnen die Mittel zur Verfügung, die Sie dazu benötigen.

Neben den bereits bekannten Fähigkeiten von Turbo Pascal, wie schnelles Compilieren, wiederverwendbare Units und objektorientierte Programmierung

unterstützt Turbo Pascal für Windows die Erzeugung und Verwendung von dynamischen Linkbibliotheken (DLLs), den direkten Zugriff auf Windows-Ressourcendateien, mehrere neue Datentypen und überarbeitete Versionen der Standard-DOS-Units von Turbo Pascal für Windows.

Programmieren für Windows in Windows

Turbo Pascal für Windows verfügt über eine integrierte Entwicklungsumgebung (IDE), die als Windows-Programm in Windows läuft. Es stehen Ihnen somit beim Schreiben, Compilieren und Testen Ihrer Turbo-Pascal-Anwendungen alle Multitasking-Fähigkeiten von Microsoft Windows zur Verfügung.

Der Ressourceneditor

Im Lieferumfang von Turbo Pascal für Windows ist neben dem Microsoft-Ressourcencompiler auch ein Ressourcen-Editor enthalten. Dieser ermöglicht die Bearbeitung von Ressourcen, wie zum Beispiel von Menüs, Dialogfenstern, Bitmaps, Symbolen und Mauszeigern, ohne Programmieraufwand. Mit dem Ressourcencompiler können die Ressourcenmakros anderer Windows-Programmierer compiliert werden.

Die ObjectWindows-Bibliothek

Bei der Programmierung für Windows werden Sie mit einigen Neuheiten konfrontiert. Dazu gehört zum Beispiel:

- die Arbeit mit Text und Grafiken in größenvariablen Fenstern,
- die Interaktion mit anderen Programmen in einer Multitasking-Umgebung und
- die Manipulation von fast 600 Funktionen der Windows-API.

Bei der Programmierung einer Anwendung, die unter Windows korrekt ausgeführt werden soll, müssen einige grundlegende Aspekte beachtet werden. Beispielsweise ist eine direkte Ausgabe auf den Bildschirm oder eine direkte Speichermodifikation nicht möglich. Das Programm muß wissen, wie es auf Botschaften reagieren soll, die Windows als Antwort auf Benutzereingaben (zum Beispiel auf eine Menüauswahl) sendet.

Die ObjectWindows-Bibliothek soll Ihnen den Einstieg in die Windows-Programmierung erleichtern. ObjectWindows ist eine objektorientierte Bibliothek, in der die meisten Elemente und Voraussetzungen für Windows-Fenster enthalten sind. Sie können die Windows-relevanten Merkmale aus der Bibliothek übernehmen und auf diesem Grundgerüst Ihr Programm aufbauen.

Turbo Debugger für Windows

Turbo Debugger für Windows ist ein leistungsfähiger Debugger auf Quelltextebene, speziell für den Einsatz mit Turbo Pascal für Windows. Er unterstützt die Auswertung von Turbo-Pascal- und Assembler-Ausdrücken und hat spezielle Fähigkeiten zum Testen von Windows-Anwendungen und objektorientierten Programmen. Eine ausführliche Beschreibung des Turbo Debugger finden Sie in Kapitel 8 »Die Kommandozeilenversion des Compilers« des *Benutzerhandbuches*.

Notwendige Vorkenntnisse

Wenn Sie Anwendungen für Windows schreiben wollen, sollten Sie mit der Programmierung in Turbo Pascal *und* mit Windows vertraut sein. Eine detaillierte Beschreibung der Elemente von Turbo Pascal finden Sie im *Benutzerhandbuch* und im *Progammierhandbuch*. Näheres zu Windows erfahren Sie in der zugehörigen Dokumentation.

Für den Gebrauch von ObjectWindows benötigen Sie gute Kenntnisse in der objektorientierten Programmierung. Mit ObjectWindows geschriebene Anwendungen machen von objektorientierten Techniken wie Vererbung und Polymorphie umfassenden Gebrauch. Näheres dazu finden Sie in Kapitel 4 »Objektorientiert programmieren« im *Benutzerhandbuch*.

Neben den objektorientierten Techniken müssen Sie mit dem Gebrauch von Zeigern und dynamischen Variablen vertraut sein, da fast alle Objektinstanzen von ObjectWindows dynamisch auf dem Heap angelegt werden.

Der Inhalt dieses Handbuchs

Wie bereits erwähnt, stellt die Verwendung von Turbo Pascal zur Windows-Programmierung eine völlige Neuheit dar. Das vorliegende Handbuch enthält deshalb umfassende Erläuterungen zu diesem Thema. Eine Referenz zu ObjectWindows und zur Windows-API finden Sie im *Windows-Referenzhandbuch*.

Das vorliegende Handbuch gliedert sich in vier große Abschnitte:

Teil 1 »Eine Einführung in ObjectWindows« macht Sie mit den Konzepten der Windows-Programmierung mit Turbo Pascal vertraut. Hier finden Sie auch ein Tutorial, das Sie Schritt für Schritt in die Entwicklung und Erweiterung einer ObjectWindows-Anwendung einführt.

Teil 2 »Arbeiten mit ObjectWindows« geht detailliert auf die Elemente von ObjectWindows ein. Sie finden hier einen Überblick über die Objekthierarchie und ihre Wechselbeziehung mit der Windows-Umgebung. Die Bestandteile der Hierarchie und ihr Einsatz werden ausführlich erläutert.

Teil 3 »ObjectWindows für Fortgeschrittene« behandelt komplexere Bereiche der Programmierung für Windows. Hier wird speziell auf Themen eingegangen, die die direkte Interaktion mit der Windows-Umgebung betreffen. Dazu gehören zum Beispiel die Arbeit mit Grafiken, die Verwendung von Ressourcen, die Speicherverwaltung und die gemeinsame Nutzung von Code und Daten. Dieser Handbuchteil enthält außerdem grundsätzliche Regeln, die bei der Windows-Programmierung beachtet werden müssen.

Teil 4 »Kollektionen und Streams« beschreibt den Einsatz der Kollektion- und Stream-Mechanismen für Objekte von Turbo Pascal, wobei besonderes Gewicht auf deren Verwendung in ObjectWindows gelegt wird.

I. Eine Einführung in Object-Windows

1. Die Vererbung des Fensters

Das vorliegende Kapitel enthält einen Überblick über die Progammierung für Microsoft Windows mit Turbo Pascal, wobei der objektorientierten Programmierung besonderes Gewicht zufällt. Die enthaltenen Beispiele basieren auf ObjectWindows, das im Lieferumfang von Turbo Pascal für Windows enthalten ist.

Sie erfahren in diesem Kapitel, woraus eine Windows-Anwendung besteht, wie sie sich verhalten muß und wie durch objektorientiertes Programmieren mit ObjectWindows Arbeitsabläufe automatisiert und vereinfacht werden können.

Für das Verständnis der folgenden Ausführungen sind Vorkenntnisse im objektorientierten Programmieren erforderlich. Sollten Sie damit noch keine Erfahrung haben, lesen Sie bitte Kapitel 4 »Objektorientiert programmieren« im *Benutzerhandbuch*. Weiterhin sollten Sie mit der Bedienung von Windows vertraut sein.

Was ist eine Windows-Anwendung?

Abbildung 1.1 zeigt die Hauptkomponenten einer Windows-Anwendung. Die Kenntnis dieser Komponenten ist Voraussetzung für das Verständnis des vorliegenden Handbuchs.

Bei einer Windows-Anwendung handelt es sich um einen speziellen Typ eines DOS-Programms mit folgenden Eigenschaften:

- Es muß in einem ausführbaren Dateiformat vorliegen (.EXE-Datei).
- Es kann nur unter Windows ausgeführt werden.
- Es läuft im allgemeinen in einem rechteckigen Bildschirmfenster ab.
- Es reagiert auf Eingaben des Benutzers in einer standardisierten Art und Weise.
- Es kann gleichzeitig mit anderen Windows-Anwendungen oder sonstigen Programmen – einschließlich weiterer Instanzen von sich selbst – ablaufen.
- Es kann mit anderen Windows-Anwendungen kommunizieren und Daten gemeinsam nutzen.

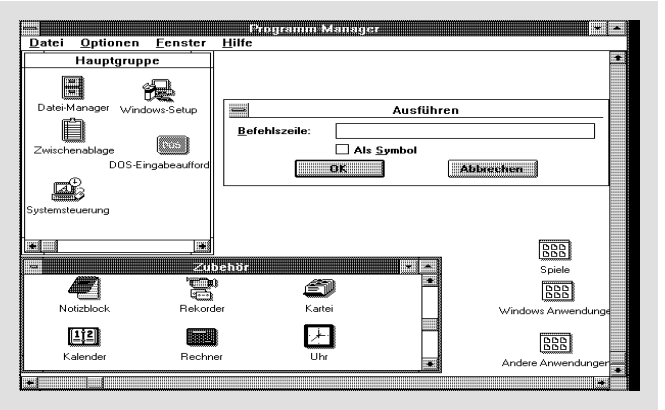


Abbildung 1.1 Die Bildschirmkomponenten einer Windows-An-

Vorteile von Windows

Windows bietet sowohl Anwendern als auch Programmierern viele Vorteile. Für den Anwender bringt es folgenden Nutzen:

- Standardisierte und konsistente Programmoberfläche.
- Das Setup für Peripheriegeräte entfällt, da Windows Gerätetreiber zur Verfügung stellt.
- Kooperation und Kommunikation zwischen verschiedenen Anwendungen.
- Multitasking, d.h. die Möglichkeit zur gleichzeitigen Ausführung mehrerer Anwendungen.
- Größere Speicherkapazität: Windows unterstützt den *Protected Mode*.

Der Programmierer hat folgende Vorteile:

- Geräteunabhängige Grafik; Grafikanwendungen können mit allen Standard-Grafikkarten ausgeführt werden.
- Unterstützung einer Vielzahl von Druckern, Bildschirmen und Mäusen.
- Eine umfangreiche Bibliothek mit Grafikroutinen.
- Mehr Speicher für größere Programme.
- Unterstützung von Menüs, Symbolen, Bitmaps etc.

Anforderungen an die Hardware

Den vielen Vorteilen von Windows stehen gewisse Anforderungen an die Hardware gegenüber. Windows setzt einen höheren Standard hinsichtlich der Grafikfähigkeiten und der Speicherkapazität voraus und erfordert einen schnel-

leren Prozessor, um dieselbe Ausführungsgeschwindigkeit wie bei einer DOS-Anwendung zu erreichen. Mindestvoraussetzung für den Einsatz von Windows ist ein 80286-Computer mit wenigstens 2 MByte Arbeitsspeicher.

Programmieren für Windows

Windows verfügt über viele Fähigkeiten, die dem Programmierer die Arbeit erleichtern.

Ereignisgesteuerte Architektur

Windows basiert auf einer ereignisgesteuerten Architektur, d.h. jede Eingabe des Anwenders wird als *Ereignis* betrachtet. Gleichgültig, ob es sich um ein Klicken mit der Maus oder das Drücken einer Taste auf der Tastatur handelt, in beiden Fällen haben wir es mit einem Ereignis zu tun, für das Windows eine Botschaft erzeugt. Drückt der Anwender zum Beispiel die linke Maustaste, erzeugt Windows eine Botschaft namens *wm_LButtonDown*. Drückt er eine Taste auf der Tastatur, wird die Botschaft *wm_KeyDown* erzeugt. Alle Menübefehle, gleichgültig, ob sie der Anwender über die Tastatur oder die Maus anwählt, werden in *wm_Command*-Botschaften umgesetzt. Die ereignisgesteuerte Architektur von Windows bildet eine optimale Voraussetzung für den Pascal-Programmierer, die sich ausgezeichnet mit dem objektorientierten Ansatz von Turbo Pascal vereinbaren läßt. Ausführliche Erläuterungen zur Erzeugung von Botschaften finden Sie in Kapitel 7 »Überblick über ObjectWindows«.

Geräteunabhängige Grafik

Windows kombiniert die Ausgabe auf den Bildschirm und auf den Drucker in einem Modul, der sogenannten grafischen Geräteschnittstelle (GDI). Diese Schnittstelle steht jedem Windows-Programm zur Verfügung. Außerdem verfügt Windows über Gerätetreiber für fast alle gebräuchlichen Grafikadapter und Drucker. Damit ist die Entwicklung einer Anwendung möglich, die ohne Änderungen auf den meisten Geräten lauffähig ist.

Geräteunabhängige Grafik beinhaltet weitere Vorteile, die nicht sofort ins Auge fallen. Zum Beispiel wird die Installation von Windows-Anwendungen sehr vereinfacht, da diese nicht mit speziellen Gerätetreibern auf das System abgestimmt werden müssen. Zudem laufen Windows-Anwendungen meist besser in einem lokalen Netz, da für jeden Anwender eine eigene lokale Konfiguration zur Verfügung steht. Geräteunabhängige Grafik stellt aber auch gewisse Anforderungen an den Programmierer, der sich an die teilweise relativ restriktiven GDI-Bestimmungen halten muß.

Multi-tasking

Windows ermöglicht es dem Anwender, mehrere Programme gleichzeitig auszuführen. Programme, die nach der Beendigung im Speicher bleiben (TSR-Programme) sind deshalb nicht mehr erforderlich. Windows verfügt aber nicht nur über Multitasking-Fähigkeiten, sondern unterstützt zusätzlich Möglichkeiten zur Kommunikation, wie zum Beispiel die Zwischenablage und den dynamischen Datenaustausch zwischen Prozessen (DDE). Ausführliche Informationen zur Kommunikation zwischen Prozessen finden Sie in Kapitel 16 »Dynamischer Datenaustausch«.

Windows verwaltet mehrere Anwendungen, indem es jeder Anwendung nur einen Teil des gesamten Bildschirms zuweist. Dabei handelt es sich um einen oder mehrere rechteckige Bereich(e), die sogenannten *Fenster*. Diese können verschoben, vergrößert, verkleinert und temporär (als Symbol) verborgen werden. Der Anwender kann dadurch sehr schnell zwischen Programmen umschalten. Für den Programmierer heißt dies, daß Text und Grafik nicht direkt an bestimmte Bildschirmpositionen geschrieben werden dürfen. Es darf nur in den sogenannten *Client-Bereich* geschrieben werden, also in den Bereich innerhalb der Fensterumrahmung. Außerdem ist zu beachten, daß sich die Anwendung den verfügbaren Arbeitsspeicher mit anderen Programmen teilen muß. Generell gilt: Es müssen die Richtlinien von Windows bezüglich Bildschirm- und Speicherverwaltung beachtet werden, damit die Anwendung korrekt ausgeführt wird.

Speicher-verwaltung

In einer typischen Windows-Arbeitssitzung werden häufig mehrere Programme gestartet und beendet. Es wäre aber wenig sinnvoll, eine Anwendung einfach nach Beendigung der vorhergehenden in den Speicher zu laden: Die Speicherkapazität wäre auf diese Weise bald erschöpft. Windows verlagert stattdessen den größten Teil einer Anwendung entweder in einen anderen Bereich des Arbeitsspeichers oder auf die Festplatte/Diskette, um für andere Programme oder für sich selbst Platz zu schaffen.

Bei der Programmierung einer Windows-Anwendung muß die dynamische Speicherverwaltung von Windows berücksichtigt werden, ein direkter Bezug auf bestimmte Speicheradressen ist nicht erlaubt. Beispielsweise würde ein traditioneller Zeiger auf eine Speicheradresse ungültig werden, wenn Windows den Speicher neu belegt, da der Zeiger immer noch auf eine Adresse zeigt, die bereits anderweitig genutzt wird. Näheres über die Verwaltung des Speichers durch Windows finden Sie in Kapitel 14 »Speicherverwaltung«.

Anstelle von Zeigern verwendet eine Windows-Anwendung zur Speicheradressierung sogenannte *Handles*, die im Grunde Zeiger auf Zeiger sind. Handles sind Zahlen, die als Index in eine von Windows verwaltete Tabelle mit Zeigern dienen. Eine Windows-Anwendung bezieht sich also mit ihrem Handle auf ein Fenster oder einen Bildschirmkontext (den Bereich des Bildschirms, in den

geschrieben wird). Windows setzt Handles außerdem für Anwendungsinstanzen, Strings, Zeichenwerkzeuge und Ressourcen, wie zum Beispiel Menüs und Symbole, ein.

Normalerweise brauchen Sie sich um die Speicher-Handles nicht selbst zu kümmern. Heap-Speicher kann mit den Routinen *New*, *Dispose*, *GetMem* und *FreeMem* reserviert und freigegeben werden. Turbo Pascal erfährt von Windows, worauf die Zeiger tatsächlich zeigen.

Einer der wichtigsten Vorteile der Speicherverwaltung von Windows ist die Fähigkeit, mehrere Anwendungen gemeinsam auf denselben compilierten Code zugreifen zu lassen. Arbeitet der Anwender beispielsweise mit zwei Instanzen desselben Programms, greifen die »beiden« Anwendungen auf denselben compilierten Code im Speicher zu. Darüber hinaus besteht die Möglichkeit, dynamische Linkbibliotheken (DLLs) zu nutzen: ein Programm kann ein Bibliotheksmodul in den Speicher laden, das auch von anderen Anwendungen genutzt werden kann.

Ressourcen

Ressourcen sind Beschreibungen der Schnittstellenmerkmale einer Windows-Anwendung: Menüs, Dialogfenster, Mauszeiger, Symbole, Bitmaps, Accelerator-Tasten und Strings. In Windows besteht die Möglichkeit, derartige Beschreibungen außerhalb des Quelltextes einer Anwendung zu verwalten. Die Ressourcen werden vor der Ausführung einer Anwendung mit der compilierten .EXE-Datei vereinigt. Eine Anwendung lädt ihre Ressourcen nur in den Speicher, wenn sie benötigt werden. Auf diese Weise wird sowenig Speicherplatz wie möglich belegt.

Die Trennung der Ressourcen-Spezifikation vom Quelltext hat einen weiteren Vorteil: Die Oberfläche einer Anwendung kann ohne Modifikation des Quelltextes geändert werden. Es ist sogar möglich, die Programmressourcen zu ändern, ohne daß der Quelltext zur Verfügung steht. Existierende Windows-Anwendungen können deshalb ohne großen Aufwand angepaßt oder übersetzt werden.

Für die Erstellung und Anpassung von Ressourcen bietet Turbo Pascal für Windows einen speziellen Ressourcen-Editor.

Dynamisches Linken

Unter Windows können Anwendungen – auch Programme in Turbo Pascal – Bibliotheksmodule während der Laufzeit ein- und auslagern. Diese Module müssen in einem speziellen ausführbaren Format vorliegen, einer sogenannten *Dynamischen Linkbibliothek (DLL)*.

Dynamische Linkbibliotheken dienen meist zur Ausführung komplexer Aufgaben, wie beispielsweise der Konvertierung eines Dateiformats. In diesem Fall kann eine Anwendung DLLs als Filter für den Dateiimport und -export verwenden. DLLs können von mehreren Anwendungen gemeinsam genutzt werden,

was eine Einsparung von Speicherplatz zur Folge hat. DLLs können mit Turbo Pascal geschrieben werden.

Zwischenablage

Mit Hilfe der Zwischenablage von Windows kann der Anwender Informationen wie zum Beispiel Text, Grafiken und Daten, in andere Teile derselben Anwendung oder in andere Anwendungen übernehmen, oder sie temporär für eine spätere Verwendung zwischenspeichern. Beispielsweise kann ein Textprogramm die Zwischenablage zum Löschen, Kopieren und Einfügen nutzen.

Dynamischer Datenaustausch

Eine weitere Möglichkeit für den Informationstransfer bildet der dynamische Datenaustausch (DDE). Während bei der Zwischenablage der Anwender eine direkte Einflußmöglichkeit hat, läuft der dynamische Datenaustausch programmgesteuert ab. Der Informationsaustausch zwischen Anwendungen erfolgt durch das Senden von DDE-Botschaften.

Multidokumentschnittstelle

Unter einer *Multidokumentschnittstelle* (MDI) versteht man eine Konvention für die Benutzerschnittstelle, die es dem Anwender ermöglicht, Fenster mit weiteren untergeordneten Fenstern zu öffnen. Die integrierte Entwicklungsumgebung (IDE) von Turbo Pascal ist ein gutes Beispiel für eine Multidokumentschnittstelle. Der Anwender kann mehrere Editierfenster öffnen, wobei jedes Editierfenster ein untergeordnetes Fenster ist.

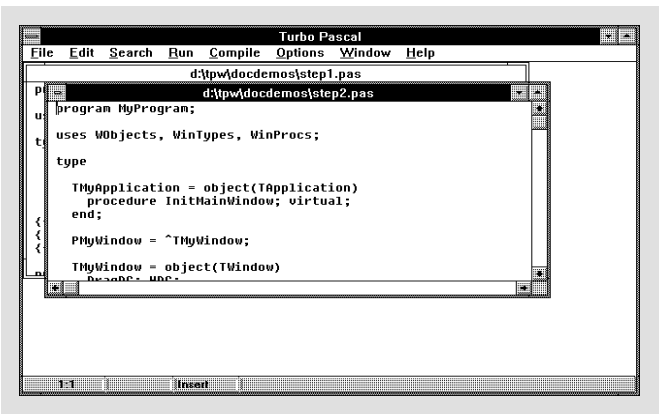


Abbildung 1.2 Die IDE von Turbo Pascal – eine MDI-Anwendung.

Datentypen von Windows

Aufgrund des Datenverwaltungsschemas von Windows und dessen enge Verknüpfung mit der Programmiersprache C sind zur Programmierung für Windows in Turbo Pascal spezialisierte Datentypen erforderlich. Beispielsweise wird ein Handle auf ein Fenster als *HWnd*-Typ gespeichert. Turbo Pascal und ObjectWindows definieren neue Pascal-Typen, die den jeweilen C-Typen entsprechen. Diese neuen Typen und Datenstrukturen werden im *Windows-Referenzhandbuch* ausführlich beschrieben.

Objektorientierte Programmierung für Windows

Die Entwicklung einer fensterorientierten Umgebung stellt hohe Ansprüche an den Programmierer, da er eine Vielzahl von Ereignissen, Formaten, Handles und anderen Anwendungen berücksichtigen muß. Sie sollten sich dadurch aber nicht entmutigen lassen. Die objektorientierte Programmierung erleichtert diese Aufgabe gewaltig, da sie es dem Anwendungsentwickler erlaubt, sich auf die Funktion des Programms zu konzentrieren. Die Form des Programms ist erst in zweiter Linie für ihn wichtig. Durch die Verwendung von Objekten zur Darstellung komplexer Strukturen (wie zum Beispiel Fenster) kann ein Turbo-Pascal-Programm Operationen und Daten kapseln – und genau das ist das Ziel von ObjectWindows.

Mit dem objektorientierten Programmieren (OOP) hat der Programmierer einen Rahmen zur Verfügung, in dem er mit Hilfe von Objekten die komplexen Schnittstellenelemente einer Windows-Anwendung darstellen kann. Bei einem Fenster handelt es sich um nichts anderes als ein Objekt. Das ObjectWindows-Fenster und die Objekttypen übernehmen den botschaftsverarbeitenden Teil einer Windows-Anwendung und erleichtern so die Interaktion zwischen Programmierer und Anwender. Die Objekte von ObjectWindows sind weitaus mehr als einfache Fenster: Sie repräsentieren Dialogfenster und Steuerelemente (zum Beispiel Listen und Schalter).

Eine bessere Schnittstelle zu Windows

ObjectWindows ermöglicht mit Hilfe der objektorientierten Erweiterungen von Turbo Pascal die Verwendung und Kapselung der Windows-Schnittstelle für die Anwendungsentwicklung (API). Dies erleichtert die Arbeit des Programmierers wesentlich, da er sich nicht mit den Details der Windows-Programmierung befassen muß. ObjectWindows stellt dem Turbo-Pascal-Programmierer drei hilfreiche Werkzeuge zur Verfügung: die Kapselung von Fensterinformationen, die Verwendung von Funktionen der Windows-API und die Beantwortung von Botschaften.

Schnittstellenobjekte

ObjectWindows definiert zwar Objekte für Fenster, Dialogfenster und Steuerelemente, stellt dabei aber nur das Verhalten des Objekts, seine Eigenschaften und die Art der Datenspeicherung zur Verfügung. Die physikalische Implementierung, d.h. die sichtbare Darstellung des Objekts auf dem Bildschirm, übernimmt Windows selbst. Die Objekte von ObjectWindows – wir nennen sie *Schnittstellenobjekte* – stehen deshalb in enger Beziehung mit den entsprechenden sichtbaren Elementen, den sogenannten *Schnittstellenelementen*. Die Herstellung der korrekten Beziehung zwischen Objekt und Element bildet den Schlüssel zur erfolgreichen Windows-Programmierung mit ObjectWindows.

Das Fenster-Handle ist Verbindungsglied in der Beziehung zwischen Objekt und Element. Beim Konstruieren eines Schnittstellenobjekts weist es Windows unter anderem an, ein Schnittstellenelement zu erzeugen. Windows gibt seinerseits ein Handle zurück, das das Element bezeichnet. Das Objekt speichert dieses Handle in einem Feld namens *HWindow*. Viele Windows-Funktionen benötigen das Fenster-Handle als Parameter – die Speicherung des Handle in einem Feld macht es für das Fensterobjekt verfügbar. Felder mit Schnittstellenobjekten können außerdem zum Speichern von Zeichenwerkzeugen oder von Statusinformationen für ein spezielles Fenster benutzt werden.

Die Abstraktion von Windows-Funktionen

Eine Windows-Anwendung bestimmt ihre Oberfläche und ihr Verhalten durch den Aufruf von Windows-Funktionen. Dafür stehen fast 600 Funktionen der Windows-API zur Verfügung. Jede Funktion akzeptiert eine Vielzahl von Parametern verschiedenster Art. Die Funktionen könnten zwar auch direkt von Turbo Pascal aufgerufen werden, ObjectWindows vereinfacht diese Aufgabe wesentlich, indem es Objektmethoden zur Abstraktion der Funktionsaufrufe zur Verfügung stellt.

Wie erwähnt, sind viele der Parameter für Windows-Funktionen bereits in den Feldern der Schnittstellenobjekte gespeichert. Die Methoden können diese Daten als Parameter für Windows-Funktionen verwenden. Zusätzlich gruppiert ObjectWindows verwandte Funktionsaufrufe zu einzelnen Methoden, die dann komplexere Aufgaben ausführen. Das Ergebnis ist eine einfach zu bedienende API, die die vorhandene Windows-API erweitert.

Dieser Ansatz reduziert zwar die Abhängigkeit des Programmierers von der Vielzahl der Windows-API-Funktionen, verhindert aber nicht den direkten Zugriff auf die API. ObjectWindows bietet somit den Vorteil einer objektorientierten Programmierung *und* der maximalen Kontrolle über die grafische Umgebung.

Beantworten von Botschaften

Die meisten Anwendungen müssen außer der Weitergabe von Anweisungen an Windows die Fähigkeit besitzen, auf die vielen Windows-Botschaften zu antworten, die sich aus Benutzereingaben (zum Beispiel einem Mausklick), anderen Programmen oder sonstigen Quellen ergeben. Die korrekte Bearbeitung

und Antwort auf derartige Botschaften ist für die einwandfreie Funktion des Programms unerläßlich. Es ist zwar relativ einfach, ein Programm zu veranlassen, auf eine Menüauswahl in einer bestimmten Weise zu reagieren und auf eine Botschaft zu antworten. Ein Programm zu schreiben, das auf annähernd 200 verschiedene Windows-Botschaften antworten kann, ist aber genau so schwierig, wie die richtigen Windows-Funktionen aufzurufen.

Objekte mit ihrem vordefinierten Verhalten (ihren Methoden) sind optimal für die Beantwortung eingehender Stimuli (den Windows-Botschaften) geeignet. ObjectWindows empfängt die Windows-Botschaften und konvertiert sie in Methodenaufrufe von Turbo Pascal. Mit Hilfe von ObjectWindows kann also eine Methode zur Beantwortung jeder Botschaft definiert werden, die von der Anwendung empfangen wird. Drückt der Anwender beispielsweise die linke Maustaste, erzeugt Windows eine Botschaft namens *wm_LButtonDown*. Soll ein Fenster oder ein Steuerelement der Anwendung auf derartige Mausklicks reagieren, kann dazu eine *WMLButtonDown*-Methode definiert werden, die auf die *wm_LButtonDown*-Botschaft antwortet. Das Objekt ruft dann die definierte Methode auf, wenn Windows diese Meldung übergibt.

Derartige Methoden werden als *Botschaftsantwort-Methoden* bezeichnet. Ohne ObjectWindows und die objektorientierte Programmierung müßte für jedes Fenster und Steuerelement eine umfangreiche case-Anweisung geschrieben werden, die überprüft, ob eine Botschaft eingegangen ist, feststellt, um welche Botschaft es sich handelt und dann über die weitere Vorgehensweise entscheidet. Mit ObjectWindows ersparen Sie sich dieses umständliche Vorgehen.

Die Struktur einer Windows-Anwendung

Bei der Vielzahl der Software-Elemente (DOS, Windows und andere Anwendungen), die miteinander in Wechselwirkung stehen, ist es hilfreich zu wissen, wie die Bestandteile einer Windows-Anwendung mit ihrer »Umgebung« zusammenarbeiten. Dieser Abschnitt befaßt sich mit der Struktur von Windows und von typischen Windows-Anwendungen, die mit Turbo Pascal unter Einsatz von ObjectWindows entwickelt wurden.

Die Struktur von Windows

Während der Ausführung befinden sich die Funktionen von Windows und der zugehörigen API in drei externen Bibliotheksmodulen, auf die bei Bedarf zugegriffen wird.

Windows verfügt über folgende Module:

KERNEL.EXE ist verantwortlich für die Verwaltung des Speichers und der Ressourcen, für den Ablauf und für die Interaktion mit DOS.

GDI.EXE steuert die Ausgabe von Grafiken auf den Bildschirm und den Drucker.

USER.EXE übernimmt die Verwaltung der Fenster und die Kommunikation mit dem Anwender.

Diese Module sind Bestandteil von Windows, befinden sich also bereits auf den Rechnern der Anwender. Das Programm, das diese Module nutzt, müssen Sie als Programmierer zur Verfügung stellen.

Interaktion mit Windows und DOS

Aufgrund der Grenzen, die dem Betriebssystem DOS gesetzt sind, ist es relativ einfach, den Anteil von DOS an der erfolgreichen Ausführung von DOS-Programmen einzuschätzen. Ein DOS-Programm läuft aufgrund der Interaktion zwischen dem Programm und den Funktionen des Betriebssystems. Dies gilt auch für Windows-Programme. Da aber Windows über weit mehr Betriebssystem-Funktionen verfügt als DOS, ist die Zusammenarbeit zwischen Windows und einer Anwendung schwieriger zu überblicken. Um zum Beispiel eine Grafik auf dem Bildschirm darzustellen, muß die Anwendung eine GDI-Funktion von Windows aufrufen. Für die Antwort auf eine Benutzereingabe (zum Beispiel einen Mausklick) muß das Programm eine Botschaftsantwort-Methode definieren. Das Programm muß also kontinuierlich mit dem Betriebssystem (DOS *und* Windows) zusammenarbeiten.

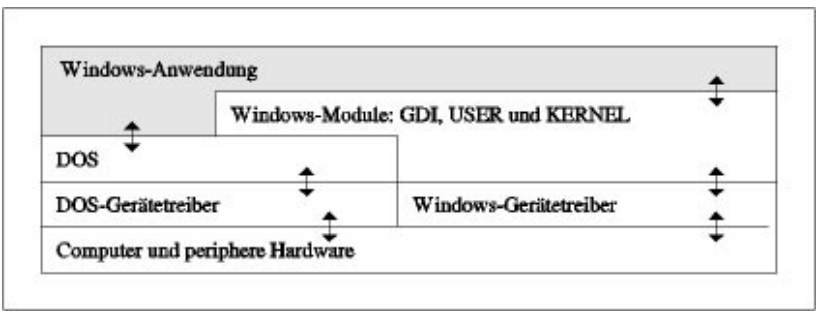


Abbildung 1.3 Interaktion von Windows-Anwendungen mit Windows und DOS. Der schattierte Bereich stellt das von Ihnen geschriebene Programm dar.

»Hello, Windows«

Eine neue Sprache oder Umgebung wird traditionellerweise mit dem Programm »Hello, World« eingeführt, das in der betreffenden Sprache bzw. für die entsprechende Umgebung geschrieben ist. Dieses Programm beschränkt sich normalerweise auf den Code, der für die Anzeige des Strings »Hello, World« auf dem Bildschirm erforderlich ist.

Für Windows reicht dies natürlich nicht aus. Zunächst muß ein Fenster eingerichtet und in dieses Fenster geschrieben werden. Anschließend wird dem Fenster mitgeteilt, wie es mit seiner Umgebung agieren bzw. auf sie reagieren soll (zumindest muß das Fenster wieder geschlossen werden können). Bereits für diese grundlegenden Schritte ist ein umfangreiches Programm erforderlich. Das Programm GENERIC.PAS (es befindet sich auf den Lieferdisketten) führt diese minimalen Schritte aus und ist bereits über 100 Zeilen lang!

Der Grund für diesen Umfang liegt in den Anforderungen, die Windows an ein Programm stellt (sogar das einfachste Programm hat einen beträchtlichen Umfang). Glücklicherweise erfüllen mit ObjectWindows geschriebene Programme die meisten dieser Anforderungen automatisch, unter anderem übernimmt ObjectWindows die Erzeugung und Anzeige des Hauptfensters und das Speichern eines Handle auf die Anwendung. Das Programm »Hello World« läßt sich dadurch auf 16 Zeilen verkürzen:

```
program Hello;

uses WObjects, WinTypes, WinProcs;

type
  THelloApp = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

procedure THelloApp.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'Hello, Turbo Pascal World'));
end;

var
  HelloApp: THelloApp;

begin
  HelloApp.Init('HelloWorld');
  HelloApp.Run;
  HelloApp.Done;
end.
```

Der Start einer Anwendung

Der erste Schritt nach dem Start eines ObjectWindows-Programms besteht in der Übernahme von vier Werten von Windows und deren Speicherung in globalen Variablen. Dies müßte »von Hand« geschehen, wenn ObjectWindows nicht zur Verfügung stünde. Bei den globalen Variablen handelt es sich um folgende:

HInstance speichert ein Handle auf die Anwendungsinstanz.

HPrevInst speichert ein Handle auf die letzte Instanz derselben Anwendung. Es ist 0, wenn es sich um die erste Instanz handelt.

CmdShow speichert einen Integer, der den Anzeigemodus für das initiale Hauptfenster repräsentiert. Diese Variable wird für Aufrufe der *Show*-Methode benötigt.

CmdLine speichert einen String mit dem Kommandozeileninhalt zum Start der Anwendung (einschließlich Optionen und Dateinamen). *CmdLine* könnte zum Beispiel '*CALC.EXE /M*' oder '*WORDPROC.EXE LETTER.DOC*' enthalten.

Die ObjectWindows-Anwendung *HelloApp* muß ein Objekt für das Hauptfenster konstruieren und initialisieren. Sie kann entweder mit *InitApplication* nur die erste Instanz von *HelloApp* initialisieren, oder sie kann jede Instanz von *HelloApp* mit der *InitInstance*-Methode initialisieren. *HelloApp* startet die Botschaftsschleife durch den Aufruf von *Run*. Sie beendet sich schließlich selbst durch die Freigabe des Anwendungsobjekts mit Hilfe der *Done*-Methode.

Das Hauptfenster

Als Hauptfenster einer Anwendung bezeichnet man das Fenster, das nach dem Start der Anwendung angezeigt wird. Es stellt dem Anwender eine Liste mit den verfügbaren Befehlen – also ein Menü – zur Verfügung. Im Verlauf der Arbeitssitzung verwaltet das Hauptfenster die Schnittstelle der Anwendung und dient in vielen Fällen als alleinige Arbeitsfläche des Programms, in dem die erforderlichen Dialogfenster eingeblendet werden. Bei komplexeren Anwendungen kann die Arbeitsfläche aus mehreren Fenstern bestehen. Schließt der Anwender das Hauptfenster, initiiert dieses den Vorgang zum Verlassen des Programms.

Vorgehensweise bei der Anwendungsentwicklung

Da jede Windows-Anwendung bestimmte Anforderungen erfüllen muß (beispielsweise die Initialisierung des Hauptfensters), nimmt man am besten eine existierende Anwendung, und paßt diese an die eigenen Bedürfnisse an. ObjectWindows verfügt über mehrere Beispielprogramme, von denen Sie das für Sie passende auswählen können.

Durch die Verwendung der integrierten Entwicklungsumgebung innerhalb von Windows läßt sich der Entwicklungsvorgang stark rationalisieren. Aufgrund der Multitasking-Fähigkeit von Windows können die IDE, Ressourcen-Editoren und die Anwendung gleichzeitig ablaufen. Turbo Pascal für Windows verfügt über Werkzeuge, die die einzelnen Aufgaben bei der Anwendungsentwicklung nicht nur erleichtern, sondern außerdem auch ihre Anzahl verringern. Die Entwicklung einer Windows-Anwendung läßt sich im Grunde auf folgende Schritte reduzieren:

1. Erzeugung des Quelltextes.

2. Erstellung der Ressourcen für Dialogfenster.

3. Compilierung des Programms (Ressourcen können automatisch dazugelinkt werden).

4. Interaktives Austesten des Programms.

Die Umgebung von Turbo Pascal für Windows wird Ihnen die Entwicklung von Windows-Programmen wesentlich erleichtern.

2. Schritt für Schritt durch Windows

Nach dieser kurzen Einführung in die ObjectWindows-Bibliothek können Sie mit der Erstellung einfacher ObjectWindows-Programme beginnen. Die folgenden Kapitel enthalten Übungen, die Sie mit der Programmierung eines grafischen, interaktiven Windows-Programm vertraut machen sollen. Dazu gehört

- die Erstellung von Menüs,
- das Speichern und Laden von Dateien,
- die Ausgabe von Grafiken und Text und
- das Anlegen eines einfachen Hilfesystems.

Sie erfahren außerdem Näheres über die wichtigsten Konzepte, die bei der Entwicklung einer Windows-Anwendung zu beachten sind, zum Beispiel die Verarbeitung von Botschaften, die Beziehungen zwischen über- und untergeordneten Fenstern und das Neuzeichnen von Grafiken.

In den Übungen werden folgende Arbeitsschritte durchgeführt:

1. Erstellen einer Anwendung.
2. Definieren eines Objekttyps für das Hauptfenster.
3. Eingeben von Text in ein Fenster.
4. Zeichnen von Linien in einem Fenster.
5. Ändern der Liniendicke.
6. Neuzeichnen der Fenster.
7. Hinzufügen eines Menüs.
8. Hinzufügen von Pop-up- und Dialogfenstern.
9. Speichern einer Zeichnung in einer Datei.
10. Hinzufügen von Steuerelementen in ein Fenster und die Verwendung von Pascal-Units.

Der Quelltext für die Anwendung ist in verschiedenen Entwicklungsstufen auf Ihren Disketten enthalten. Die Dateien haben die Namen STEP1.PAS, STEP2.PAS etc., je nachdem, welcher Schritt (englisch step) gerade ausgeführt wird.

Abbildung 2.1 zeigt, wie die Anwendung nach der Durchführung aller Arbeitsschritte aussieht.

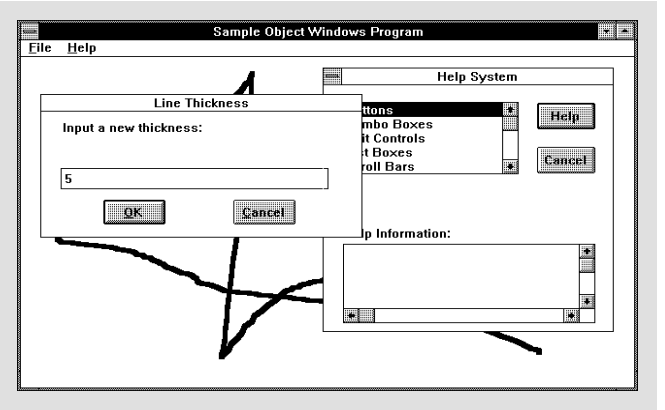


Abbildung 2.1 Eine Object Windows-Anwendung

Schritt 1: Eine einfache Windows-Anwendung

Sie beginnen Ihre Arbeit mit dem Schreiben einer äußerst einfachen Object-Windows-Anwendung mit dem Namen *MyProgram*.

Dieses Programm, es ist in der Datei STEP01.PAS enthalten, können Sie als Basis für alle Programme verwenden, die Sie mit ObjectWindows schreiben.

Die Anwendung *MyProgram* instantiiert und erzeugt das Hauptfenster der Anwendung.

Voraussetzungen

Alle Windows-Programme verfügen über ein Hauptfenster, das angezeigt wird, wenn der Anwender das Programm startet. Das Verlassen des Programms erfolgt durch das Schließen des Hauptfensters. In einer ObjectWindows-Anwendung ist das Hauptfenster ein *Fensterobjekt*, das dem *Anwendungsobjekt* zugehörig ist. Das Anwendungsobjekt ist für die Erzeugung und das Anzeigen des Hauptfensters, die Bearbeitung von Windows-Botschaften und die Beendigung der Anwendung verantwortlich. Es agiert sozusagen als objektorientierter Ersatz für die Anwendung selbst.

ObjectWindows stellt in gleicher Weise Objekttypen für Fenster, Dialogfenster etc. zur Verfügung, die Ihnen die Details der Windows-Programmierung ersparen.

Jedes ObjectWindows-Programm muß einen neuen Anwendungstyp definieren, der vom vorhandenen Typ, *TApplication*, abgeleitet ist. In *MyProgram* heißt dieser Typ *TMyApplication*. Der Rumpf von *MyProgram* sieht folgendermaßen aus:

```
var
  MyApp: TMyApplication;
begin
  MyApp.Init('MyProgram');
  MyApp.Run;
  MyApp.Done;
end.
```

Init ist der Konstruktor von *TMyApplication* und erzeugt ein neues Anwendungsobjekt mit dem Namen *MyApp*. Außerdem setzt er den Namen der Anwendung (ein Objektfeld) auf 'Myprogram', erzeugt das Hauptfenster und zeigt es an. *Run* löst eine Reihe von Funktionsaufrufen aus, die die neue Windows-Anwendung starten. *Done* ist der Destruktor von *TMyApplication*.

Definieren des Anwendungstyps

Ihre Anwendung muß einen neuen Typ von dem Standard-ObjectWindows-Typ *TApplication* (oder von einem von *TApplication* abgeleiteten Typ) ableiten. Dieser neue Typ muß mindestens eine geerbte Methode – *InitMainWindow* – überschreiben. *TApplication.InitMainWindow* ist eine abstrakte Methode, die von ObjectWindows automatisch zum Einrichten des Hauptfensters aufgerufen wird. Jede ObjectWindows-Anwendung *muß* ihr eigenes Hauptfenster konstruieren.

Die Definition des Typs *TMyApplication* sieht folgendermaßen aus:

```
TMyApplication = object(TApplication)
  procedure InitMainWindow; virtual;
end;
```

InitMainWindow ist für die Konstruktion des Fensterobjekts verantwortlich, das als Hauptfenster der Anwendung dient. Das Hauptfensterobjekt wird im *MainWindow*-Feld des Anwendungsobjekts gespeichert. Zwar gehört das Hauptfensterobjekt zum Anwendungsobjekt, die beiden Objekte sind aber nicht hierarchisch miteinander verknüpft. Das Verhältnis der beiden Objekte zueinander wird als Instanzenverbindung bezeichnet.

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil,
    'Sample ObjectWindows Program'));
end;
```

Normalerweise werden Sie die obige Methode modifizieren, um einen neuen Hauptfenstertyp zu erhalten. Die Methode verwendet eine Objektinstanz von *TWindow*, und zwar einen in ObjectWindows enthaltenen Fenstertyp, der ein einfaches Fenster definiert. In Schritt 2 wird diese Objektinstanz durch einen komplexeren Fenstertyp ersetzt. Abbildung 2.2 zeigt, wie *MyProgram* auf dem Bildschirm dargestellt wird.

Wenn Sie ein ObjectWindows-Programm schreiben wollen, benötigen Sie zumindest die Unit *WObjects*. Bei komplexeren Anwendungen müssen weitere Units verwendet werden. Beispielweise muß jedes Programm, das direkt auf die Windows-API zugreift, die Units *WinProcs* und *Wintypes* enthalten.

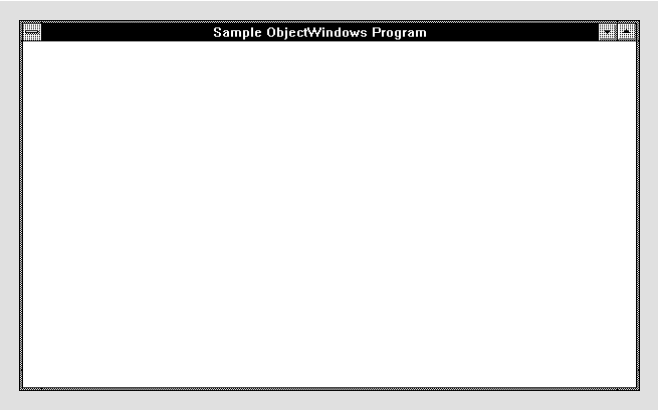


Abbildung 2.2 Ihr erstes ObjectWindows-Programm: MyProgram

Bisher erzeugt *MyProgram* nur ein leeres Fenster, das verschoben, vergrößert und verkleinert und geschlossen werden kann. Das vollständige (in STEP01.PAS enthaltene) Listing von *MyProgram* sieht momentan wie folgt aus:

```
program MyProgram;

uses WObjects;

type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil,
    'Sample ObjectWindows Program'));
end;

var MyApp: TMyApplication;
begin
  MyApp.Init('MyProgram');
  MyApp.Run;
  MyApp.Done;
end.
```

Schritt 2: Der Objekttyp für das Hauptfenster

Das mit Schritt 1 erstellte Programm besteht aus zwei Objekten: einem Anwendungsobjekt und einem Fensterobjekt. Das Anwendungsobjekt *MyApp* ist eine Instanz des Typs *TMyApplication*, der von *TApplication* abgeleitet ist. Das

Fensterobjekt, das im *MainWindow*-Feld des Anwendungsobjekts gespeichert ist, ist eine Instanz von *TWindow*, einem generischen Fenster von ObjectWindows. In fast allen (außer in ganz einfachen) Programmen müssen Sie Ihren Fenstertyp für das Hauptfenster selbst definieren, und dabei das anwendungsspezifische Verhalten einarbeiten. In diesem Schritt werden wir dem Hauptfenster »Leben einhauchen«, indem wir einen komplexeren, von *TWindow* abgeleiteten Fenstertyp definieren.

Was ist ein Fensterobjekt?

In Schritt 1 haben wir ein Anwendungsobjekt kennengelernt, das alle Standardverhaltensweisen einer Windows-Anwendung zur Verfügung stellt, einschließlich der Konstruktion des Hauptfensters. Der Typ *TApplication* enthält alle grundlegenden Verhaltensweisen und Eigenschaften, die Sie für Ihre Windows-Anwendungen benötigen.

In ähnlicher Weise stellt ein Fensterobjekt die Verhaltensweisen der Fenster zur Verfügung, die von ObjectWindows-Anwendungen erzeugt werden. Dazu gehören auch die Hauptfenster dieser Anwendungen. Unter Verhaltensweisen für Fenster verstehen wir deren Anzeige, das Vergrößern und Verkleinern, die Reaktion auf Benutzereingaben (wie Klicken und Bewegen mit der Maus, Auswahl von Menüpunkten etc.) und die Anzeige von Steuerelementen, wie Listen und Schalter. Der Typ *TWindow* und sein Vorfahre *TWindowsObject* stellen Ihnen die grundlegenden Methoden und Objektfelder für diese Verhaltensweisen zur Verfügung. Um Ihre Programme so zu gestalten, daß sie für den Anwender transparent und leicht zu bedienen sind, müssen Sie neue, von *TWindow* abgeleitete Fenstertypen erzeugen. Diese neuen Typen erben die Methoden und Felder von *TWindow* und fügen neue hinzu. Der objektorientierte Ansatz erspart Ihnen also die ständige »Neudefinition des Fensters«.

Handles

Alle Fensterobjekte haben mindestens drei Felder: *HWindow*, *Parent* und *ChildList*. Wie bereits in Kapitel 1 »Die Vererbung des Fensters« erwähnt, enthält *HWindow* das Handle auf das Fenster. Ein Handle ist eine Nummer, die ein Schnittstellenobjekt (zum Beispiel ein Fenster, ein Dialogfenster oder ein Steuerelement) mit seinem entsprechenden Schnittstellenelement verknüpft.

HWindow enthält also einen Integer, der ein Schnittstellenelement bezeichnet. Man könnte das Handle mit der Nummernmarke vergleichen, die Sie bei der Abgabe eines Kleidungsstücks an der Garderobe erhalten. So, wie Sie durch die Rückgabe dieser Marke Ihr Kleidungsstück erhalten, erhalten Sie mit dem Handle das zugehörige Fenster. Bei der Arbeit mit Fensterobjekten ist eine direkte Änderung des Fenster-Handle normalerweise nicht erforderlich. Es wird aber benötigt, um eine Windows-Funktion direkt aufzurufen. Um zum Beispiel ein Meldungsfenster anzuzeigen, brauchen Sie nur die Funktion *MessageBox* aufzurufen. *MessageBox* erfordert die Angabe eines Parameters, der das übergeordnete Fenster des Meldungsfensters bezeichnet. In unserem Beispiel geben wir das Hauptfenster an, dessen Handle im *HWindow*-Objektfeld enthalten ist:

```
MessageBox(MainWindow^.HWindow, 'Do you want to save?', 'File has
changed,' mb_YesNo or mb_IconQuestion);
```

Übergeordnete und untergeordnete Fenster

Die meisten Fenster einer Anwendung sind mit anderen Fenstern verknüpft. Wird zum Beispiel eine Anwendung beendet, muß sie alle Fenster entfernen, die sie angelegt hat. Windows teilt die Fenster in übergeordnete (Eltern) und untergeordnete Fenster (Kinder) ein. Ein übergeordnetes Fenster ist für seine untergeordneten Fenster verantwortlich. ObjectWindows stellt für jedes Fensterobjekt Felder zur Verfügung, um die »Überwachung« der übergeordneten und deren untergeordnete Fenster zu ermöglichen.

Das *Parent*-Feld enthält das Objekt des übergeordneten Fensters. Es handelt sich dabei weniger um ein Verwandtschafts- als um ein Zugehörigkeitsverhältnis. Diese Beziehungen zwischen den Fenstern werden in Schritt 8 beschrieben.

ChildList, das dritte Fensterobjekt-Feld, enthält eine verkettete Liste mit den untergeordneten Fenstern (falls derartige Fenster vorhanden sind). Wir werden in Schritt 8 untergeordnete Fenster hinzufügen.

Erstellen eines neuen Fenstertyps

Wir wissen nun, was in einem Fensterobjekt enthalten ist, und können einen neuen Fenstertyp entwerfen. Er stammt vom Typ *TWindow* ab und dient als Hauptfenster für *MyProgram*. Dazu aktualisieren wir zuerst die Typdefinitionen für den neuen Typ *TMyWindow*. Außerdem wird ein Zeiger auf den Typ, *PMyWindow*, definiert, der bei der Instantiierung von *TMyWindow* hilfreich ist.

```
type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
  end;
```

Anschließend aktualisieren wir *TMyApplication.InitMainWindow*, damit ein *TMyWindow* (und nicht ein *TWindow*) als Hauptfenster erzeugt wird.

```
procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow, Init (nil,
    'Sample ObjectWindows Program'));
end;
```

Für die Definition eines neuen Typs für das Hauptfenster von *TMyProgram* reicht es aus, den Typ zu definieren und ihn in *InitMainWindow* zu instantiieren. Das Anwendungsobjekt ruft Methoden zur Erzeugung des Schnittstellenelements für das Fenster (*Create*) und zu dessen Anzeige auf dem Bildschirm (*Show*) auf. Sie werden diese Methoden kaum direkt einsetzen, sie werden normalerweise beim Aufruf der *MakeWindow*-Methode des Anwendungsobjekts automatisch aufgerufen (*MakeWindow* wird in Kapitel 10 »Fensterobjekte« ausführlich besprochen).

Beachten Sie, daß *TMyWindow* keine neuen Verhaltensweisen definiert, sondern nur über diejenigen von *TWindow* und *TWindowsObject* verfügt. Mit anderen Worten, unser Programm *MyProgram* wird dadurch nicht interessanter.

Im nächsten Schritt werden wir nun einige neue Verhaltensweisen hinzufügen.

Beantworten von Botschaften

Einem Fensterobjekt läßt sich am einfachsten eine sinnvolle Aufgabe zuweisen, indem man festlegt, wie es auf Windows-Botschaften zu antworten hat. Klickt der Anwender beispielsweise mit der linken Maustaste im Hauptfenster von *MyProgram*, erhält das entsprechende Fensterobjekt von Windows eine *wm_LButtonDown*-Botschaft. Diese teilt dem Fensterobjekt mit, daß sein Bereich vom Anwender mit der Maus angeklickt wurde. Außerdem übergibt sie die Koordinate des Punkts, an dem der Mausklick stattfand. (Diese Informationen werden in Schritt 6 verwendet.)

Ähnliches liegt vor, wenn der Anwender mit der rechten Maustaste klickt. Das Hauptfensterobjekt erhält dann von Windows die Botschaft *wm_RButtonDown*. Unsere Aufgabe ist es nun, dem Hauptfenster – einer Instanz von *TMyWindow* – mitzuteilen, wie es auf diese Botschaften reagieren soll.

Windows-Botschaften werden abgefangen und beantwortet, indem man für jeden Typ eingehender (relevanter) Botschaften eine Methode definiert. Diese Methoden werden als Botschaftsantwort-Methoden bezeichnet. Um den Kopf einer Methodendefinition als Botschaftsantwort zu kennzeichnen, fügen wir eine Erweiterung hinzu. Dabei handelt es sich um den Bezeichner der Botschaft, auf die geantwortet werden soll.

Die Methode, die wir als nächstes definieren, antwortet auf alle *wm_LButtonDown*-Botschaften:

```
TMyWindow = object(TWindow)
procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
    wm_LButtonDown;
end;
```

Msg ist ein *TMessage*-Record, der beispielsweise die Koordinaten des angeklickten Punktes enthält. Wir befassen uns in Schritt 3 ausführlicher mit dem Parameter *Msg*.

Im Moment beschränken wir uns auf Antwortmethoden, die auf ein Klicken mit der Maus durch die Anzeige eines Meldungsfensters reagieren. Später werden wir noch weitere zweckmäßigere Antworten hinzufügen. Die Definition für die Antwortmethode auf den Druck der linken Maustaste lautet wie folgt:

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the left mouse button',
    'Message Dispatched', mb_OK);
end;
```

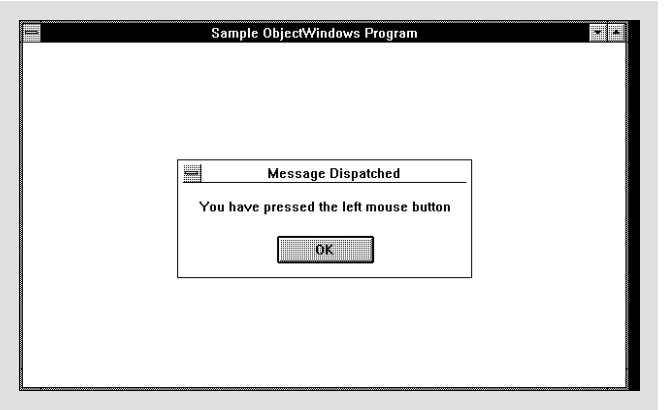


Abbildung 2.3 Die Antwort von MyProgram auf eine Benutzereingabe

Den kompletten Quelltext für diesen Schritt finden Sie am Ende dieses Kapitels.

Beenden eines Programms

Unser Programm wird beendet, wenn der Anwender das Systemmenüsymbol im Hauptfenster – das kleine Quadrat in der oberen linken Ecke – doppelt anklickt. Das Fenster wird daraufhin sofort geschlossen und die Anwendung beendet. Diese Verhaltensweise eignet sich zwar gut für einfache Programme, ist aber nicht in allen Fällen erwünscht. Ein gutes Programm sollte den Anwender beim Verlassen des Programms immer fragen, ob er seine Arbeit speichern will (falls dies noch nicht geschehen ist). Sie können Ihre ObjectWindows-Anwendungen ohne großen Aufwand mit dieser Funktion versehen. Wir wollen nun unser Programm *MyProgram* mit einer derartigen Sicherung versehen.

Versucht der Anwender, ein ObjectWindows-Programm zu verlassen, wird die virtuelle *CanClose*-Methode der Anwendung aufgerufen. *CanClose* ist eine Funktion vom Typ Boolean, die anzeigt, ob das Beenden des Programms zulässig ist (*True*). Die von *TApplication* geerbte *CanClose*-Methode ruft standardmäßig die *CanClose*-Methode des Hauptfensterobjekts auf. In fast allen Fällen bestimmt das Hauptfensterobjekt, ob die Beendigung des Programms zulässig ist.

Unser Hauptfenstertyp *TMyWindow* erbt von *TWindowsObject* eine *CanClose*-Methode, die wiederum für jedes seiner untergeordneten Fenster (falls vorhanden) eine *CanClose*-Methode aufruft. Gibt es keine untergeordneten Fenster (in unserem Programm ist das der Fall), liefert *CanClose True* zurück. Um das Schließverhalten einer Anwendung zu ändern, definieren wir eine neue *CanClose*-Methode für den Objekttyp des Hauptfensters:

```
function TMyWindow.CanClose: Boolean;
var Reply: Integer;
begin
  CanClose := True;
```

```
    Reply := MessageBox(HWindow, 'Do you want to save?', 'File has
      changed', mb_YesNo or mb_IconQuestion);
    if Reply = idYes then CanClose := False;
  end;
```

Beim Versuch, *MyProgram* zu beenden, erhält der Anwender jetzt ein Meldungsfenster mit der Frage »Do you want to save?«. Anklicken des Yes-Schalters ergibt *False* und verhindert, daß das Hauptfenster geschlossen und das Programm beendet wird. Anklicken des No-Schalters ergibt *True* und beendet das Programm. Abbildung 2.4 zeigt das modifizierte *MyProgram*.

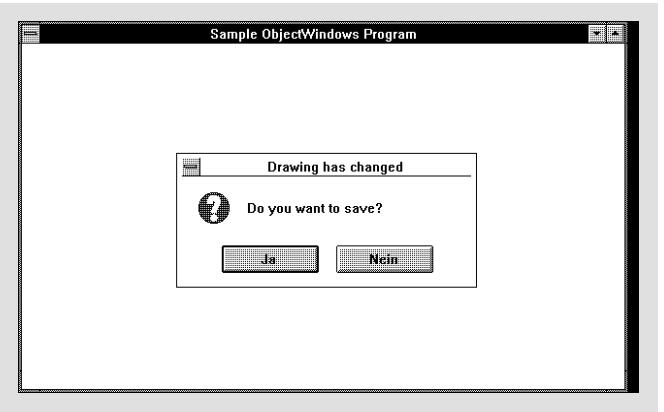


Abbildung 2.4 MyProgram mit besserem Schließverhalten

Der vollständige (in STEP02.PAS enthaltene) Quelltext von *MyProgram* lautet bis jetzt:

```
program MyProgram;

uses WinTypes, WinProcs, WObjects;

type
  TMyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
end;

type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    function CanClose: Boolean; virtual;
    procedure WMLButtonDown(var Msg: TMessage);
      virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
      virtual wm_First + wm_RButtonDown;
end;

function TMyWindow.CanClose: Boolean;
var
```

```
  Reply: Integer;
begin
  CanClose := True;
  Reply := MessageBox(HWindow, 'Do you want to save?',
    'Drawing has changed', mb_YesNo or mb_IconQuestion);
  if Reply = id_Yes then CanClose := False;
end;

procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the left mouse button',
    'Message Dispatched', mb_OK);
end;

procedure TMyWindow.WMRButtonDown(var Msg: TMessage);
begin
  MessageBox(HWindow, 'You have pressed the right mouse button',
    'Message Dispatched', mb_OK);
end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow, Init(nil,
    'Sample ObjectWindows Program'));
end;

var
  MyApp: TMyApplication;

begin
  MyApp.Init('MyProgram');
  MyApp.Run;
  MyApp.Done;
end.
```

3. Füllen eines Fensters

Für die Verwaltung der Benutzerschnittstelle unseres Programms haben wir nun Menüs, Dialog- und Pop-up-Fenster zur Verfügung. Trotzdem läßt sich mit dem Programm kaum etwas Sinnvolles anfangen. Im folgenden wollen wir *MyProgram*, das eigentlich erst der Rahmen für ein Programm ist, in eine brauchbare, interaktive Grafikanwendung verwandeln. Dazu ermöglichen wir dem Anwender zunächst das Zeichnen von Text und von Linien im Hauptfenster von *MyProgram*. Anschießend sorgen wir dafür, daß ein Neuzeichnen der Grafiken möglich ist, daß die Dicke von Linien geändert werden kann und daß Grafiken als Datei gespeichert werden können.

Was ist ein Bildschirmkontext?

Ein Bildschirmkontext repräsentiert die Zeichenoberfläche eines Fensters. Windows benötigt den Bildschirmkontext zum Zeichnen von Text oder Grafiken in einem Fenster. Die Verwaltung des Bildschirmkontexts erfolgt im Speicherbereich von Windows, die Anwendung kann aber über ein Handle darauf zugreifen. Ein Handle auf einen Bildschirmkontext ist wie ein Fenster-Handle eine Zahl, die einen bestimmten Bildschirmkontext bezeichnet.

Hinweis: Näheres über Gerätekontexte finden Sie in Kapitel 17 »Die grafische Geräteschnittstelle«.

Zum Zeichnen in einem Fenster ist ein Bildschirmkontext erforderlich, daher erzeugen wir ein neues Objektfeld für das Hauptfensterobjekt. Es hat den Namen *DragDC* und enthält das Handle auf einen Bildschirmkontext. *DragDC* ist vom Typ *HDC*, einem in Windows definierten Typ, der dem Standardtyp Word in Turbo Pascal entspricht.

Um einen Bildschirmkontext zu erhalten, rufen wir unmittelbar vor dem Zeichnen auf dem Bildschirm in einer der Fenstertyp-Methoden die Windows-Funktion *GetDC* auf:

```
DragDC := GetDC(HWindow);
```

DragDC kann nun als Parameter in Aufrufen von Windows-Grafikfunktionen eingesetzt werden, die ein Handle auf einen Bildschirmkontext benötigen. Dazu zwei Beispiele:

```
TextOut(DragDC, 20, 20, 'Sample Text', 11);
LineTo(DragDC, 30, 45);
```

Wenn das Zeichnen von Text oder Grafik beendet ist, *muß* der Bildschirmkontext freigegeben werden:

```
ReleaseDC(HWindow, DragDC);
```

Die Freigabe der Bildschirmkontexte nach dem Zeichnen ist unbedingt erforderlich, da die Windows-Umgebung nur über fünf Bildschirmkontexte verfügt, die sonst innerhalb kurzer Zeit aufgebraucht wären. Dies hätte eine fehlerhafte Programmausführung (in den meisten Fällen einen Absturz) zur Folge. Stürzt Ihre Anwendung beim dritten oder vierten Zeichenvorgang ab, sollten Sie sie zuerst auf nicht-freigegebene Bildschirmkontexte überprüfen.

Der Bildschirmkontext dient zur Unterstützung wichtiger Zeichenfunktionen. Er stellt sicher, daß Text und Grafiken nicht außerhalb der Fensteroberfläche gezeichnet werden, und er verwaltet die Auswahl und das Löschen von Zeichenwerkzeugen wie Stifte, Pinsel und Schriften. Die Auswahl eines neuen Stiftes werden wir später beim Zeichnen einer Grafik durchführen. Vorerst wollen wir uns mit dem Zeichnen von Text begnügen.

Schritt 3: Zeichnen von Text in einem Fenster

Um Text in das Hauptfenster von *MyProgram* zu zeichnen, benötigen wir einen Bildschirmkontext. Dazu führen wir die Schritte zur Anforderung und späteren Freigabe des Kontexts aus (vgl. den vorherigen Abschnitt). Um das Ganze interessanter zu gestalten, zeichnen wir den Text als Reaktion auf den Druck der linken Maustaste (diese war in Schritt 2 die Anzeige eines Dialogfensters). Die Antwort soll aus dem Zeichnen von Text bestehen, der die Positionskoordinaten des Mausklicks angibt. Im Koordinatensystem von Windows definiert z.B. (20,30) einen Punkt, der 20 Pixel rechts vom linken und 30 Pixel unter dem oberen Rand der Zeichenoberfläche des Fensters liegt.

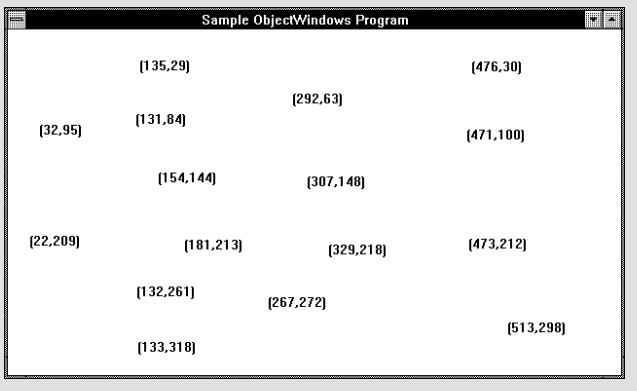


Abbildung 3.1 MyProgramm zeichnet Text an einem angeklickten Punkt

Beachten Sie, daß das Klicken mit der linken Maustaste eine *wm_LButtonDown*-Botschaft erzeugt, die mit der Botschaftsantwort-Methode *WMLButtonDown* abgefangen wird.

Botschafts-Records

In Schritt 2 haben wir gesehen, daß das *Msg*-Argument einer Botschaftsantwort-Methode wertvolle Informationen (wie zum Beispiel die Koordinaten eines angeklickten Punktes) enthalten kann. *Msg* ist ein *TMessage*-Record mit den Feldern zur Aufnahme des Longint-Parameters *lParam* und des Word-Parameters *wParam*. Die Bezeichner *lParam* und *wParam* haben entsprechende Felder in der Botschaftsstruktur von Windows, *TMsg*.

TMessage definiert außerdem Felder zur Aufnahme der Unterfelder von *lParam* und *wParam*. Beispielsweise enthält *Msg.lParamLo* das niederwertige und *Msg.lParamHi* das höherwertige Word von *lParam*. Meist haben wir es mit folgenden Feldern zu tun: *wParam*, *lParamLo* und *lParamHi*.

Im Fall von *WMLButtonDown* enthält *Msg.lParamLo* die X-Koordinate des angeklickten Punkts, während *Msg.lParamHi* die Y-Koordinate speichert. Wir wollen nun *WMLButtonDown* so schreiben, daß es einen Text mit den Koordinaten des angeklickten Punkts zeichnet. Dazu müssen *Msg.lParamLo* und *Msg.lParamHi* in Strings konvertiert und diese mit zwei Klammern und einem Komma verkettet werden. Das Ergebnis könnte zum Beispiel so aussehen: '(25,34)'. In unserem Beispiel verwenden wir zur Formatierung des Strings die Windows-Funktion *WVSPrintF* (*WVSPrintF* wird in Kapitel 2 »Windows-Funktionen« im *Windows-Referenzhandbuch* ausführlich besprochen).

Damit der String an den angeklickten Punkt gezeichnet wird, werden an Windows in einem *TextOut*-Funktionsaufruf der String *S*, die Koordinaten *Msg.lParamLo* und *Msg.lParamHi* und die Stringlänge *StrLen(S)* übergeben. Vor dem Zeichnen muß ein Bildschirmkontext angefordert und hinterher wieder freigegeben werden.

Hinweis: Windows erwartet, daß nullterminierte Strings übergeben werden (d.h. die Strings müssen mit einem Null-Byte enden). In Kapitel 13 »MDI-Objekte« wird ausführlich auf diese Strings eingegangen.

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
var
  DC: HDC;
  S: array[0..9] of Char;
begin
  WVSPrintF(S, '(%d,%d)', Msg.LParam);
  DC := GetDC(HWindow)
  TextOut(DC, Msg.LParamLo, Msg.LParamHi, S, StrLen(S));
  ReleaseDC (HWindow, DC);
end;
```

Der Aufruf von *WVSPrintF* übernimmt die Typumwandlung und Formatierung.

Löschen des Bildschirms

Unser Programm benötigt nun noch eine Funktion zum Löschen des Bildschirms. Es ist möglich, gezeichneten Text durch das Vergrößern oder Verkleinern oder durch das Verbergen und Wiedersichtbarmachen des Fensters zu löschen. Es ist aber sinnvoller, dem Anwender das Löschen des Bildschirms über eine Menüauswahl oder eine andere Tätigkeit (zum Beispiel Klicken mit der Maus) zu ermöglichen. In unserem Programm soll der Bildschirm durch das Drücken der rechten Maustaste gelöscht werden. Wir erreichen dies, indem wir die *wmRButtonDown*-Methode so umdefinieren, daß sie die Windows-Prozedur *InvalidateRect* aufruft, die ein Neuzeichnen des gesamten Fensters bewirkt.

Da unser Fenster noch nicht weiß, wie es sich selbst neu zeichnen muß, löscht es nur seinen Client-Bereich.

```
procedure TMyWindow.WMRButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
end;
```

Der momentane Stand von *MyProgram* ist in der Datei STEP03.PAS enthalten.

Schritt 4: Zeichnen von Linien im Fenster

Nachdem wir die Grundlagen für das Zeichnen auf dem Bildschirm kennen (Anfordern eines Bildschirmkontexts, Zeichnen und Freigabe des Bildschirmkontexts), können wir nun eine komplexere, interaktive Grafikanwendung erstellen. Die nächsten Schritte bestehen aus der Erstellung eines einfachen Programms, das dem Anwender das Zeichnen im Hauptfenster ermöglicht. Dazu wird folgendermaßen vorgegangen:

1. Antworten auf das Drücken der linken Maustaste und das Bewegen der Maus. Die Punkte werden entlang des Wegs verbunden, woraus sich gezeichnete Linien ergeben.

2. Antworten auf das Drücken der rechten Maustaste. Es wird ein Eingabefeld angezeigt, in dem der Anwender die Liniendicke ändern kann.

3. Automatisches Neuzeichnen des Fensterinhalts. Dazu werden die Punkte gespeichert und als Antwort auf eine *paint*-Botschaft neu gezeichnet.

Bewegungen der Maus

Rufen wir uns zunächst ins Gedächtnis, wie Windows auf Mausereignisse reagiert. Wie wir bereits wissen, hat das Drücken der linken Maustaste eine

wm_LButtonDown-Botschaft und den Aufruf der *WMLButtonDown*-Methode zur Folge.

Bisher haben wir auf das Drücken der linken Maustaste durch die Anzeige von Meldungsfenstern und das Zeichnen von Text geantwortet.

Wir wissen auch, daß das Drücken der rechten Maustaste eine *wm_RButtonDown*-Botschaft und den Aufruf der *WMRButtonDown*-Methode zur Folge hat (unsere Antwort darauf war das Löschen des Bildschirms). Diese Antworten betrafen alle nur den jeweils ersten Druck auf eine Maustaste. Bei vielen interaktiven Windows-Programmen kann der Anwender aber eine Maustaste drücken und die Maus dann bewegen, um Linien oder Rechtecke zu zeichnen oder um Grafiken zu verschieben. Wir wollen nun auch in unserem Beispielprogramm Mausbewegungen abfangen und auf sie durch das Zeichnen von Linien reagieren.

Wir tun dies durch das Beantworten weiterer Botschaften. Windows liefert die Botschaft *wm_MouseMove*, wenn der Anwender die Maus verschiebt, und *wm_LButtonUp*, wenn er die linke Maustaste losläßt.

Normalerweise empfängt das Fenster eine *wm_LButtonDown*-Botschaft, darauf folgt eine Reihe von *wm_MouseMove*-Botschaften (eine für jeden Punkt, über den die Maus bewegt wird). Schließlich folgt noch eine *wm_LButtonUp*-Botschaft.

Ein typisches Windows-Grafikprogramm antwortet auf *wm_LButtonDown* durch die Initiierung eines Zeichenvorgangs (dazu gehört unter anderem auch die Anforderung eines Bildschirmkontexts). *wm_MouseMove*-Botschaften werden durch das Zeichnen oder Verschieben von Grafiken und *wm_LButtonUp* durch die Beendigung des Zeichenvorgangs (einschließlich der Freigabe des Bildschirmkontexts) beantwortet.

Die folgende Tabelle faßt die gebräuchlichsten Botschaften für Mausereignisse zusammen.

Tabelle 3.1 Botschaften für Mausereignisse

Botschaft	Ereignis
wm_LButtonDown	Anwender drückt auf die linke Maustaste.
wm_RButtonDown	Anwender drückt auf die rechte Maustaste.
wm_MouseMove	Anwender bewegt die Maus.
wm_LButtonUp	Anwender läßt die linke Maustaste los.
wm_RButtonUp	Anwender läßt die rechte Maustaste los.
wm_LButtonDblClk	Anwender führt einen Doppelkick mit der linken Maustaste aus.
wm_RButtonDblClk	Anwender führt einen Doppelkick mit der rechten Maustaste aus.

Beantworten von Mausbewegungs-Botschaften

Die folgende Tabelle zeigt, wie auf Botschaften, die das Programm nach dem Bewegen der Maus erhält, reagiert werden muß, wenn ein Programm zum Linienzeichnen erstellt werden soll.

Tabelle 3.2 Die in Schritt 4 verwendeten Botschaften

Botschaft	Antwort
wm_LButtonDown	Löschen des Bildschirm, Anfordern und Speichern eines Bildschirmkontexts in *DragDC*. Plazieren des Zeichenstifts auf den angeklickten Punkt.
wm_MouseMove	Zeichnen einer Linie vom vorherigen zum aktuellen Punkt.
wm_LButtonUp	Freigabe von *DragDC*.

Wie bereits ausgeführt, folgt auf *wm_LButtonDown* immer *wm_LButtonUp*. Dazwischen können sich (müssen aber nicht) *wm_MouseMove*-Botschaften befinden. Auf jeden Fall muß nach jeder Anforderung eines Bildschirmkontexts dieser später wieder freigegeben werden. In unserem Fall wird für alle Zeichenvorgänge, die zwischen dem Anklicken der linken Maustaste und deren Wiederloslassen liegen, nur ein einziger Bildschirmkontext verwendet.

Die Kombination einer Anforderung eines Bildschirmkontexts mit seiner Freigabe ist wichtig für die einwandfreie Funktion von Grafikprogrammen. Es ist deshalb vorteilhaft, eine weitere Sicherung einzubauen. Definieren wir dazu ein neues Boolesches Objektfeld für *TMyWindow* (dem Hauptfenstertyp) mit dem Namen *ButtonDown*. *WMLButtonDown* setzt dieses Feld auf *True*, *WMLButtonUp* setzt es auf *False*. Der Wert von *ButtonDown* kann dann vor dem Anfordern und der Freigabe des Bildschirmkontexts überprüft werden.

Die drei Methoden für die Beantwortung der durch Mausbewegungen hervorgerufen Botschaften sehen folgendermaßen aus:

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
    DragDC := GetDC(HWindow);
    MoveTo(DragDC, Msg.LParamLo, Msg.LParamHi);
  end;
end;

procedure TMyWindow.WMMouseMove(var Msg: TMessage);
begin
  if ButtonDown then
    LineTo(DragDC, Integer(Msg.LParamLo), Integer(Msg.LParamHi));
end;

procedure TMyWindow.WMLButtonUp(var Msg: TMessage);
begin
  if ButtonDown then
  begin
    ButtonDown := False;
```

```
      ReleaseCapture;
      ReleaseDC(HWindow, DragDC);
    end;
end;
```

MoveTo und *LineTo* sind Grafikfunktionen der Windows-API, die die aktuelle Zeichenposition verschieben bzw. eine Linie zur aktuellen Position zeichnen. Sie benötigen für die korrekte Ausführung ein Handle auf den Bildschirmkontext *DragDc*. Denken Sie immer daran, daß Sie nicht direkt im Fenster sondern im zugehörigen Bildschirmkontext zeichnen.

SetCapture und *ReleaseCapture* sind Windows-Funktionen, die die korrekte Übergabe der *wm_MouseMove*-Botschaften durch Windows sicherstellen. Wird die Maus zum Beispiel aus dem Fenster hinausbewegt, sendet Windows die *wm_MouseMove*-Botschaften an das Hauptfenster und nicht an das benachbarte Fenster. Die Objektdefinition für *TMyWindow* muß mit Methoden-Köpfen für *WMMouseMove* und *WMLButtonUp* aktualisiert werden:

```
procedure WMLButtonUp(var Msg: TMessage); virtual wm_First +
  wm_LButtonUp;
procedure WMMouseMove(var Msg: TMessage); virtual wm_First +
  wm_MouseMove;
```

Der momentane Stand von *MyProgram* ist in der Datei STEP04.PAS enthalten.

Schritt 5: Ändern der Liniendicke

Bis zum jetzigen Zeitpunkt können mit dem Programm *MyProgram* nur dünne Linien gezeichnet werden. Grafikprogramme ermöglichen aber normalerweise die Änderung der Liniendicke. Um unserem Programm diese Eigenschaft zu verleihen, ändern wir tatsächlich nicht die Liniendicke, sondern die Stärke des Stiftes, mit dem die Linien gezeichnet werden. Stifte gehören neben Pinseln, Schriften und Paletten zu den Zeichenwerkzeugen, die der Bildschirmkontext zur Verfügung stellt. Indem wir *MyProgram* die Fähigkeit zum Setzen einer neuen Liniendicke geben, lernen wir auch, wie einem Bildschirmkontext neue Zeichenwerkzeuge zugeordnet werden können.

Der Anwender soll die Dicke der Linien in einem Eingabefeld wählen können. Abbildung 3.2 zeigt, wie dieses Eingabefeld – es ist vom Typ *TInputDialog* – aussieht.

Steuern der Liniendicke

Für diesen Programmierschritt sind etwas detailliertere Kenntnisse der Windows-Grafik und des Bildschirmkontexts erforderlich, auf die wir kurz eingehen wollen.

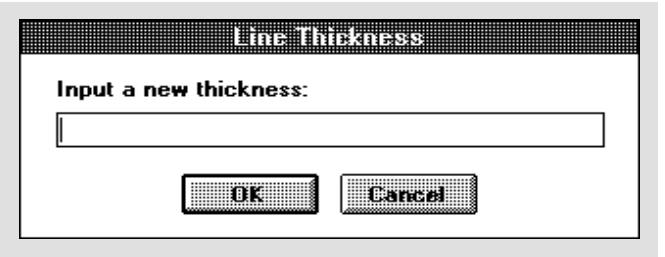


Abbildung 3.2 Ändern der Liniendicke

Die Zeichenwerkzeuge zur Erstellung von Grafiken und Text in einem Fenster wurden bereits kurz angesprochen:

- Stifte,
- Pinsel und
- Schriften.

Es handelt sich dabei um Fensterelemente, deren Beschreibungen im Speicherbereich von Windows enthalten sind (ähnlich wie bei anderen Schnittstellenelementen). Anwendungen greifen auf Zeichenwerkzeuge mit Hilfe von Handles zu.

Da ObjectWindows die Zeichenwerkzeuge nicht nachbildet, müssen die entsprechenden Objekte von den Programmen erzeugt und später wieder aus dem Windows-Speicher entfernt werden.

Ein Zeichenwerkzeug ist im Grunde nichts anderes als der Pinsel eines Malers, während der Bildschirmkontext die Leinwand darstellt. Nach dem »Anfertigen« der Zeichenwerkzeuge (Pinsel) und der Beschaffung der Leinwand (Bildschirmkontext) erfolgt die Auswahl des benötigten Zeichenwerkzeugs.

Entsprechend wird auch in einem Windows-Programm das korrekte Zeichenwerkzeug für einen Bildschirmkontext gewählt. Zu jedem Bildschirmkontext gehört ein Standardsatz von Zeichenwerkzeugen, die nicht explizit ausgewählt werden müssen: ein dünner schwarzer Stift, ein kompakter schwarzer Pinsel und eine Systemschrift. Wir benötigen jedoch für unser Beispiel einen anderen (etwas dickeren) Stift, der nicht im Standardsatz der Zeichenwerkzeuge enthalten ist.

Der erste Schritt besteht darin, *MyProgram* die Anzeige von Eingabefeldern zu ermöglichen. Dazu fügen wir zur uses-Anweisung die Unit *StdDlgs* hinzu. Außerdem benötigen wir *Strings*, damit die Windows-kompatiblen Funktionen zur Stringmanipulation verwendet werden können.

Der Anfang des Programms sieht dann folgendermaßen aus:

```
program MyProgram;
uses Strings, WinTypes, WinProcs, WObjects, StdDlgs;
...
```

Jetzt brauchen wir ein Objektfeld für *TMyWindow*, in dem ein Handle auf das benötigte Zeichenwerkzeug gespeichert werden kann. In unserem Programm beschränken wir uns hier auf das Zeichnen und Anzeigen von Linien mit jeweils nur einer Liniendicke. Der Stift, der dieser Dicke entspricht, wird in einem neuen

TMyWindow-Feld namens *ThePen* gespeichert. Außerdem wird eine Methode mit dem Namen *SetPenSize* benötigt, das die Erzeugung des neuen Stiftes und das Löschen des alten übernimmt.

Die Objektdeklaration von *TMyWindow* sieht wie folgt aus:

```
type
  PMyWindow = ^TMyWindow;
  TMyWindow = object(TWindow)
    DragDC: HDC;
    ButtonDown: Boolean;
    ThePen: HPen;
    PenSize: Integer;
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  destructor Done; virtual;
  function CanClose: Boolean; virtual;
  procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
    wm_LButtonDown;
  procedure WMLButtonUp(var Msg: TMessage); virtual wm_First +
    wm_LButtonUp;
  procedure WMMouseMove(var Msg: TMessage); virtual wm_First +
    wm_MouseMove;
  procedure WMRButtonDown(var Msg: TMessage); virtual wm_First +
    wm_RButtonDown;
  procedure SetPenSize(NewSize: Integer);
end;
```

Zur Initialisierung dieser neuen Datenfelder modifizieren wir nun *den Init*-Konstruktor, so daß er den Stift einrichtet, und überschreiben den *Done*-Destruktor, damit er den Stift freigibt.

In den neuen Methoden muß ein Aufruf der geerbten Methoden enthalten sein:

```
constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  ButtonDown := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
end;

destructor TMyWindow.Done;
begin
  DeleteObject(ThePen);
  TWindow.Done;
end;
```

Wir ändern nun die Methode *WMLButtonDown* dahingehend, daß sie den aktuellen Stift (*ThePen*) für den neu angeforderten Bildschirmkontext auswählt. Ist *ThePen* gleich 0, wurde kein spezieller Stift erzeugt und es wird der voreingestellte (dünne) Stift verwendet.

Wie *MoveTo* und *MessageBox* ist auch *SelectObject* eine Funktion der Windows-API.

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  InvalidateRect(HWindow, nil, True);
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
    DragDC := GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LParamLo, Msg.LParamHi);
  end;
end;
```

Diese Methode wählt den bereits erzeugten Stift für den Bildschirmkontext. Die tatsächliche Erzeugung des Stifts erfolgt mit der Methode *SetPenSize*:

```
procedure TMyWindow.SetPenSize(NewSize: Integer);
begin
  DeleteObject(ThePen);
  ThePen := CreatePen(ps_Solid, NewSize, 0);
  PenSize := NewSize;
end;
```

Der Aufruf der Windows-Funktion *CreatePen* ist eine Möglichkeit zur Erzeugung eines Windows-Zeichenstifts mit einer bestimmten Dicke. In *ThePen* wird ein Handle auf den Stift gespeichert.

Das Löschen des vorherigen Stiftes ist wichtig, da andernfalls der Speicher von Windows bald erschöpft wäre.

Verwendung des Eingabefeldes

Das Eingabefeld zur Änderung der Liniendicke soll durch den Druck der rechten Maustaste eingeblendet werden. Dazu müssen wir die Methode *WMRButtonDown* so umdefinieren, damit sie ein Eingabefeld anzeigt. Wir benutzen ein Eingabefeld, das in ObjectWindows vordefiniert ist. Es handelt sich um ein einfaches Dialogfenster mit einer Zeile für die Texteingabe. Für seine Verwendung ist keine Änderung von *TInputDialog* oder dessen Methoden erforderlich.

Da das Eingabefeld nur für eine kurze Zeit angezeigt wird und für seine Verwaltung nur eine Methode erforderlich ist, brauchen wir es nicht als Objektfeld von *TMyWindow* zu definieren. Es kann als lokale Variable der Methode *WMRButtonDown* existieren – die Konstruktion und Entfernung des Eingabefeldes erfolgt deshalb innerhalb der *WMRButtonDown*-Methode.

Nachdem das Objekt des Eingabefelds mit *Init* konstruiert wurde, kann es durch einen Aufruf der *Execute*-Methode als modales Dialogfenster benutzt werden. Die Verwendung der *Execute*-Methode entspricht dem Einsatz von

Create für ein Fensterobjekt: Sie erzeugt das Element, das dem Schnittstellenobjekt entspricht. Die Verarbeitung von *Execute* endet aber erst, wenn der Anwender das Dialogfenster durch Anklicken von OK oder Cancel schließt. Klickt er OK an, wird *InputText* mit der Eingabe des Anwenders gefüllt. Dies erfolgt durch einen Aufruf der *GetText*-Methode von *TInputDialog*. Da eine Nummer für die Liniendicke benötigt wird, muß der Text in eine Zahl konvertiert und in einem Aufruf von *SetPenSize* übergeben werden. Bei jeder Auswahl einer neuen Liniendicke durch den Anwender muß der alte Stift gelöscht und ein neuer erzeugt werden.

```
procedure TMyWindow.WMRButtonDown(var Msg: TMessage);
var
  InputText: array[0..5] of Char;
  NewSize, ErrorPos: Integer;
begin
  if not ButtonDown then
  begin
    Str(PenSize, InputText);
    if Application^.ExecDialog(New(PInputDialog,
      Init(@Self, 'Line Thickness', 'Input a new thickness:',
        InputText, SizeOf(InputText)))) = id_Ok then
    begin
      Val(InputText, NewSize, ErrorPos);
      if ErrorPos = 0 then SetPenSize(NewSize);
    end;
  end;
end;
```

Vor Beendigung des Programms muß noch der letzte Stift aus dem Windows-Speicher gelöscht werden. Dazu überschreiben wir den *Done*-Konstruktor:

```
destructor TMyWindow.Done;
begin
  DeleteObject(ThePen);
  TWindow.Done;
end;
```

Schritt 6: Automatische Neuanzeige von Grafiken

Wahrscheinlich sind Sie überrascht, daß Text und Grafiken, die mit den Windows-Funktionen *TextOut* und *LineTo* in ein Fenster gezeichnet wurden, beim Verkleinern, Vergrößern oder beim Sichtbarmachen des Fensters verschwinden. Warum ist das so?

Bisher wurden Text oder Linien nicht in Variablen gespeichert. Nachdem Grafikdaten über Funktionsaufrufe an Windows übergeben wurden, ist es nicht möglich, für eine Neuanzeige des Bildschirms auf sie zuzugreifen. Um den Zugriff zu ermöglichen, müssen die Grafikdaten irgendwie gespeichert werden.

Ein Objekt eignet sich vorzüglich für diese Aufgabe. Es kann einfache und komplizierte Grafiken speichern und problemlos als Objektfeld des Hauptfensters verwaltet werden.

Neuzeichnen von Fenstern

Wenn der Anwender unser Fenster in der Größe verändert oder verbirgt und wieder sichtbar macht, muß es aktualisiert (neu gezeichnet) werden. Ist ein Neuzeichnen erforderlich, ruft ObjectWindows automatisch die *Paint*-Methode für den betreffenden Fenstertyp auf. Die von *TWindow* geerbte *Paint*-Methode führt keine Funktion aus.

Sie müssen in die Methode *Paint* den Code zum Zeichnen des Fensterinhalts schreiben. Sie wird auch beim erstmaligen Anzeigen des Fensters aufgerufen. *Paint* übernimmt außerdem das Neuzeichnen des Fensters mit dem aktuellen Inhalt.

Es gibt einen großen Unterschied zwischen dem Zeichnen von Grafiken mit der *Paint*-Methode und den anderen bisher verwendeten Methoden (zum Beispiel Zeichnen als Antwort auf ein Mausereignis). *Paint* verfügt bereits über einen Bildschirmkontext. Er wird durch den Parameter *PaintDC* übergeben, das Programm braucht ihn deshalb nicht anzufordern oder freizugeben. Allerdings müssen die Zeichenwerkzeuge für *PaintDC* neu ausgewählt werden.

Für das Zeichnen des Fensterinhalts führen wir die Schritte noch einmal aus, die für die Erstellung der ursprünglichen Zeichnung in *DragDC* erforderlich waren, verwenden aber nun *PaintDC* anstelle von *DragDC*.

Das Ergebnis auf dem Bildschirm ist das gleiche, die Grafikdaten müssen aber als Objekte gespeichert werden, bevor sie in einer *Paint*-Methode verwendet werden können.

Speichern von Grafiken als Objekte

Wir wollen eine Linie als Kollektion von Punkten in einem Objektfeld von *TMyWindow* namens *Points* speichern. Dafür verwenden wir den Typ *TCollection* aus der *WObjects*-Unit. *Points* enthält die aktuelle Zeichnung zu jedem Zeitpunkt in Form einer Kollektion von Punkten. Muß das Fenster neu gezeichnet werden, werden die Daten in *Points* zur Darstellung der Linie verwendet.

Nun stellt sich folgende Frage: »Was ist eigentlich eine Linie?« Aus Schritt 4 wissen wir, daß eine gezeichnete Linie aus einer Kollektion von Punkten besteht, die Windows dem Programm über die *wm_MouseMove*-Botschaft übergibt. Für die Darstellung von Punkten wird ein Objekttyp benötigt. Für die Speicherung der X- und Y-Koordinaten als Objektfelder definieren wir *DPoint*.

Objekte von *TCollection* wachsen dynamisch mit dem Hinzufügen weiterer Elemente und eignen sich deshalb gut für das Speichern einer unbekannten Zahl von Linien und Punkten.

Die Objektdefinition für *DPoint* lautet wie folgt:

```
type
  PDPoint = ^TDPoint;
  TDPoint = object(TObject)
    X, Y: Integer;
    constructor Init(AX, AY: Integer);
  end;

constructor TDPoint.Init(AX, AY: Integer);
begin
  X := AX;
  Y := AY;
end;
```

DPoint definiert zwar keine neue Verhaltensweise, muß aber ein Objekt sein, damit es später in einem Stream gespeichert werden kann.

Points muß in *TMyWindow.Init* konstruiert sowie in *TMyWindow.Done* freigegeben werden. Dazu werden die von *TWindow* geerbten *Init* und *Done* überschrieben.

```
constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  ButtonDown := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
  Points := New(PCollection, Init(50, 50));
end;

destructor TMyWindow.Done;
begin
  Dispose(Points, Done);
  DeleteObject(ThePen);
  TWindow.Done;
end;
```

Zur Erinnerung: Das Hauptfenster von *MyProgram* enthält eine Kollektion von Punkten in seinem *Points*-Feld. Zeichnet der Anwender Linien, müssen diese in Objekte konvertiert und zu *Points* hinzugefügt werden. Wird das Fenster anschließend neu gezeichnet, muß über *Points* iteriert und jeder enthaltene Punkt neu gezeichnet werden.

Damit Linien als Objekte gespeichert werden können, müssen wir *WMLButtonDown* und *WMMouseMove* modifizieren:

```
procedure TMyWindow.WMLButtonDown(var Msg: TMessage);
begin
  Points^.FreeAll;
  InvalidateRect(HWindow, nil, True);
  if not ButtonDown then
  begin
    ButtonDown := True;
    SetCapture(HWindow);
```

```
    DragDC := GetDC(HWindow);
    SelectObject(DragDC, ThePen);
    MoveTo(DragDC, Msg.LParamLo, Msg.LParamHi);
    Points^.Insert(New(PDPoint, Init(Msg.LParamLo, Msg.LParamHi)));
  end;
end;

procedure TMyWindow.WMMouseMove(var Msg: TMessage);
begin
  if ButtonDown then
  begin
    LineTo(DragDC, Integer(Msg.LParamLo), Integer(Msg.LParamHi));
    Points^.Insert(New(PDPoint, Init(Msg.LParamLo, Msg.LParamHi)));
  end;
end;
```

WMLButtonUp muß nicht geändert werden.

Neuzeichnen gespeicherter Grafiken

TMyWindow enthält nun die aktuelle Linie und muß angewiesen werden, diese auf Befehl zu zeichnen. Der Befehl, den wir dazu verwenden, lautet *Paint*. Wir schreiben jetzt eine *Paint*-Methode für *TMyWindow*, die die Aktionen von *WMLButtonDown*, *WMMouseMove* und *WMLButtonUp* wiederholt. Durch das Iterieren über die Kollektion von Punkten erhält *Paint* die Punkte in der Reihenfolge, in der sie vom Anwender eingegeben wurden. Die *Paint*-Methode sieht wie folgt aus:

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var First: Boolean;

procedure DrawLine(P: PDPoint); far
begin
  if First then MoveTo(PaintDC, P^.X, P^.Y);
  else LineTo(PaintDC, P^.X, P^.Y);
  First := False;
end;

begin
  SelectObject(PaintDC, ThePen);
  First := True;
  Points^.ForEach(@DrawLine);
end;
```

4. Hinzufügen eines Menüs

Die meisten Windows-Anwendungen enthalten im Hauptfenster ein Menü, das dem Anwender eine Reihe von Auswahlpunkten zur Verfügung stellt (beispielsweise *Datei/Sichern*, *Datei/Öffnen* oder *Hilfe*). In diesem Kapitel werden wir für *MyProgram* ein derartiges Menü erstellen.

In einer fensterorientierten Umgebung fällt die Auswahl aus einem Menü in die gleiche Kategorie wie ein Klicken mit der Maus: In beiden Fällen handelt es sich um ein Ereignis, das vom Anwender hervorgerufen wird. Die Antwort auf die Auswahl eines Menüpunkts erfolgt im wesentlichen so wie die Beantwortung anderer Benutzereingaben. Sie umfaßt folgende Schritte:

1. Entwurf des Menüs als Menüressource.
2. Laden der Menüressource in das Hauptfensterobjekt.
3. Definition von Antworten auf Menüauswahlen.
4. Angabe der Ressourcendatei.

Menü-ressourcen

In einer Anwendung, die über ein Menü verfügt, muß eine Menüspezifikation mit dem Text der Auswahlpunkte und der Struktur der Haupt- und Unterpunkte vorhanden sein. Dies gilt für jede ObjectWindows-Anwendung, wobei die Spezifikation nicht Bestandteil des Pascal-Quelltextes der Anwendung ist, sondern separat in der sogenannten *Ressource* enthalten ist. Die Ressourcen werden von Windows in kompakter und effizienter Weise gespeichert. Für den interaktiven Entwurf von Menüs und anderen Ressourcen (beispielsweise Dialogfenster, Symbole und Bitmaps) können wir den Ressourcen-Editor verwenden.

Eine Anwendung greift auf ihre zugehörigen Ressourcen mit Hilfe einer Ressourcen-ID zu. Es kann sich dabei um einen Integerwert (zum Beispiel 100), oder eine Integerkonstante (zum Beispiel *MyMenu*) handeln. Die Unterscheidung der einzelnen Menüauswahlen erfolgt durch die Menü-ID.

Fahren wir mit unserem Programm *MyProgram* fort. Mit dem Ressourcen-Editor oder dem Ressourcen-Compiler wird eine Menüressource erzeugt und als .RES-Datei mit dem Namen COOKBOOK.RES gespeichert. Diese Datei ist auf

Ihren Disketten enthalten und liegt mit COOKBOOK.RC auch als Quelltext vor. Die .RES-Datei wird nach ihrer Erzeugung folgendermaßen in der *$R*-Compiler-Direktive angegeben:

```
{$R COOKBOOK.RES}
```

Die folgende Abbildung 4.1 zeigt dieses Menü (Ressourcen-ID 100), das Help-Menü (Menü-ID 901) und den Auswahlpunkt *File* mit den Unterpunkten *New*, *Open*, *Save* und *SaveAs* (Menü-ID 101, 102, 103 und 104). Hauptmenüpunkte, zu denen Unterpunkte existieren, haben keine Menü-IDs, ihre Auswahl hat nur die Anzeige der Unterpunkte zur Folge. Hat ein Hauptmenüpunkt keine Unterpunkte (wie beispielsweise *Help*), verfügt er über eine Menü-ID, die eine Aktion ausführen kann.

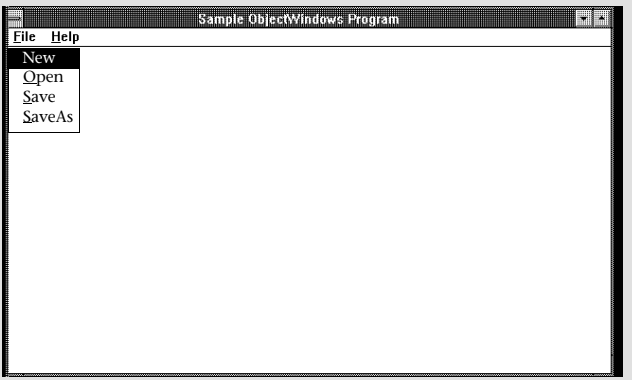


Abbildung 4.1 MyProgram mit einer Menüressource

Da das Menü außerhalb von ObjectWindows entworfen wurde und der Umgang mit dem Menü ziemlich einfach ist, muß für das Menü kein neues Objekt angelegt werden. Stattdessen wird es zu einem Attribut des Hauptfensters (wie es zum Beispiel auch der Titel des Hauptfensters ist). Im nächsten Schritt wird das Menü mit dem Hauptfenster verknüpft.

Schritt 7: Ein Menü für das Hauptfenster

Wie erwähnt, ist das Menü einer Anwendung kein separates Objekt, das dem Hauptfenster zugehörig ist. Das Menü ist überhaupt kein Objekt, sondern ein Attribut des Hauptfensters. Es wird im *Menu*-Feld von *Attr* gespeichert. *Menu*-Feld ist ein Fensterobjektfeld, das einen Record der *Erstellungsattribute* für das Fenster enthält. Zum Setzen des Menüattributs müssen wir unter anderem den *Init*-Konstruktor für unseren Fenstertyp (*TMyWindow*) umdefinieren.

Die in COOKBOOK.RES gespeicherte Menüressource hat die Ressourcen-ID 100. Wir erhalten die Ressource durch einen Aufruf der Windows-Funktion *LoadMenu*:

```
LoadMenu(HInstance, PChar(100));
```

PChar(100) konvertiert die Zahl 100 in einen Stringtyp namens PChar, einem Zeiger auf ein Zeichen-Array. Strings, die als Argumente an Windows-Funktionen übergeben werden, müssen vom Typ *PChar* sein. Für einen Zugriff auf *PChar* muß die Compiler-Direktive *$X+* eingeschaltet sein (dies ist die Voreinstellung).

Alternativ dazu kann eine Menüressource auch einen symbolischen Bezeichner wie beispielsweise 'SAMPLE_MENU' haben. In diesem Fall wird die Menüressource folgendermaßen geladen:

```
LoadMenu (HInstance, 'SAMPLE_MENU');
```

TMyWindow.Init muß folgendermaßen aussehen (beachten Sie, daß für die Initialisierung aller Fensterobjekte als erstes *TWindow.Init* aufgerufen werden muß):

```
constructor TMyWindow.Init(AParent: PWindwosObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ButtonDown := False;
  PenSize := 1;
  ThePen := CreatePen(ps_Solid, PenSize, 0);
  Points := New(PCollection, Init(50, 50));
end;
```

Wird nun das Hauptfenster angezeigt, enthält es das in Abbildung 4.1 gezeigte Menü. Allerdings zeigt die Auswahl eines Menüpunktes noch keine Wirkung. Dazu müssen wir die folgenden Schritte zum Abfangen und Beantworten der Menübotschaft ausführen.

Abfangen der Menübotschaft

Wenn der Anwender einen Menüpunkt auswählt, empfängt das mit dem Menü verknüpfte Fenster von Windows eine *Befehlsbotschaft*. Zur Verarbeitung einer dieser Botschaften definieren wir eine Methode für den Objekttyp *TMyWindow* und verwenden dabei eine spezielle Erweiterung:

```
procedure FileNew(var Msg: TMessage); virtual cm_First + 101;
```

Hierbei ist *cm_First* eine in ObjectWindows vordefinierte Konstante, die den Beginn des Bereichs von Konstanten für Befehle festlegt. *101* ist die gewünschte Menü-ID. Dieser auf *cm_First* basierende Index für dynamische Methoden ist nicht mit dem zur Beantwortung eingehender Windows-Botschaften (er basiert auf dem Offset *wm_First*) zu verwechseln. Es handelt sich hier um einen Spezialfall, der nur bei Menüs und Accelerator-Tasten verwendet wird. (Botschaftsbereiche und -Offsets werden in Kapitel 7 »Überblick über ObjectWindows« ausführlich beschrieben.)

Zur Verbesserung der Lesbarkeit ersetzen wir die Menü-IDs durch Konstanten. Sie werden am Anfang des Programms definiert:

```
const
  cm_New = 101;
  cm_Open = 102;
  cm_Save = 103;
  cm_SaveAs = 104;
  cm_Help = 901;
```

Jetzt können alle Methoden zur Beantwortung von Befehlsbotschaften definiert werden:

```
procedure FileNew(var Msg: TMessage); virtual cm_First + cm_New;
procedure FileOpen(var Msg: TMessage); virtual cm_First + cm_Open;
procedure FileSave(var Msg: TMessage); virtual cm_First + cm_Save;
procedure FileSaveAs(var Msg: TMessage); virtual cm_First +
   cm_SaveAs;
procedure Help(var Msg: TMessage); virtual cm_First + cm_Help;
```

Beantwortung der Menübotschaft

Es steht nun für jede Menüauswahl eine Methode zur Verfügung. Wählt der Anwender zum Beispiel Help, wird die *Help*-Methode aufgerufen. Momentan wird in diesem Fall nur ein Meldungsfenster eingeblendet:

```
procedure TMyWindow.Help(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Feature not implemented.', 'Help', mb_OK);
end;
```

Die folgende Abbildung zeigt die Antwort von *MyProgram* auf die Auswahl von Help.

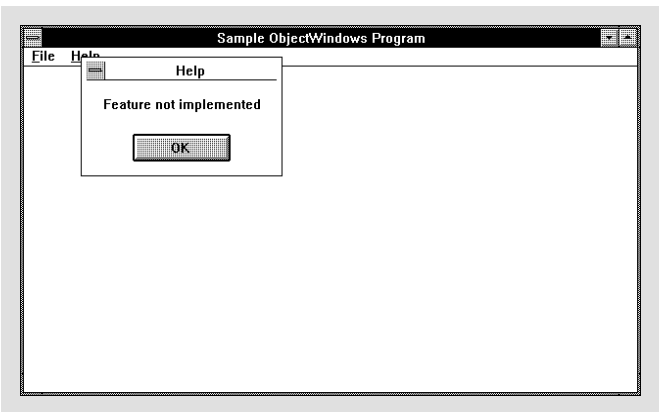


Abbildung 4.2
Das Hilfesystem von MyProgram

Damit nach der Auswahl von *File/New* der Bildschirm gelöscht wird, fügen wir folgende *FileNew*-Methode hinzu:

```
procedure TMyWindow.FileNew(var Msg: TMessage);
begin
  Points^.FreeAll;
  InvalidateRect(HWindow, nil, True);
end;
```

Diese Methode löscht alle gespeicherten Punkte und erzwingt ein Neuzeichnen des Bildschirms. Da keine Punkte zum Neuzeichnen vorhanden sind, ist das Resultat ein leerer Bildschirm.

Für *FileOpen*, *FileSave* und *FileSaveAs* können ähnliche Dummy-Methoden geschrieben werden. An späterer Stelle werden wir sie modifizieren, damit sie sinnvolle Aktionen ausführen.

Der momentane Quelltext von *MyProgram* ist in der Datei STEP7.PAS enthalten.

Die Verknüpfung von Ressourcen zu einer ausführbaren Datei

Die Verknüpfung der Menüressource mit dem ObjectWindows-Programm erfolgt nach dessen Compilierung. Die Compiler-Direktive *$R* sorgt dafür, daß dies am Ende des Compiler- und Link-Vorgangs geschieht. Die mit der Direktive angegebene Ressourcendatei wird an die compilierte .EXE-Datei angehängt.

Ressourcen können zu ausführbaren Dateien hinzugefügt oder von ihnen entfernt werden. Vorhandene Ressourcen können modifiziert werden. Näheres hierzu finden Sie im *Ressourcen-Editor*.

5. Konstruktion eines Dialogs

Bisher hatten wir es nur mit zwei Fenstertypen zu tun: mit dem Hauptfenster (dem *TMyWindow*-Objekt) und mit Meldungsfenstern (zur Anzeige von »Feature not implemented« und »Are you sure you want to quit?«). Ein richtiges Windows-Programm verfügt aber meist über viele verknüpfte Fensterobjekte (Fenster, Steuerelemente und Dialogfenster). Mit Ausnahme des Hauptfensters hat jedes Fenster ein übergeordnetes Fenster, von dem es abhängig ist. Diese abhängigen Fenster werden untergordnete Fenster genannt. Wenn der Anwender beispielsweise ein übergeordnetes Fenster schließt, werden auch dessen untergeordnete Fenster geschlossen.

Aus diesen Beziehungen ergibt sich für jede Anwendung ein Netz von übergeordneten und untergeordneten Fenstern, wobei jeweils das Hauptfenster als oberstes übergeordnetes Fenster fungiert. Die meisten Fenster sind einem Fenster untergeordnet und haben ihrerseits wieder untergeordnete Fenster. Waisen, d.h. völlig unabhängige Fenster gibt es nicht.

Die folgende Abbildung zeigt die – bisher nur teilweise implementierten – Beziehungen zwischen übergeordneten und untergeordneten Fenstern von *MyProgram*.

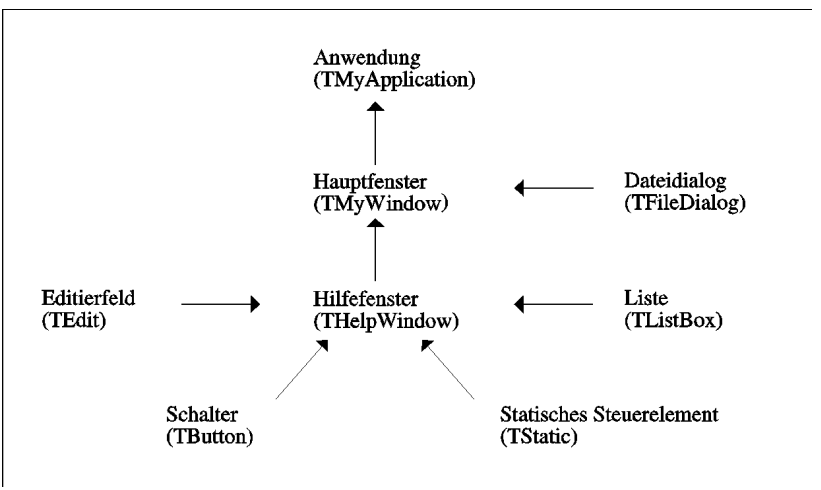


Abbildung 5.1 Eine Gruppe von über- und untergeordneten Fenstern

Untergeordnete Fenster lassen sich in zwei Gruppen einteilen, selbständige und unselbständige untergeordnete Fenster. Selbständige untergeordnete Fenster, zu denen beispielsweise Meldungs- und Pop-up-Fenster gehören, steuern ihre Anzeige und ihre Position selbst. Unselbständige untergeordnete Fenster haben eine feste Position im übergeordneten Fenster und werden mit diesem verschoben. Steuerelemente, wie zum Beispiel Schalter in Meldungsfenstern, gehören zu diesem Fenstertyp. Wir werden im folgenden Schritt zwei Typen von selbständigen untergeordneten Fensterobjekten erstellen: Pop-up-Fenster und Dialogfenster.

Über- und untergeordnete Fenster dürfen nicht mit Vorfahren und abgeleiteten Typen verwechselt werden. Für über- und untergeordnete Fenster ist das Zugehörigkeitsverhältnis relevant. Beispielsweise gehört das Hilfefenster (das wir im folgenden konstruieren werden) zum Hauptfenster. Diese Art der Zugehörigkeit in Windows entspricht in etwa der Instanzen-Verbindung der objektorientierten Programmierung.

Wir berücksichtigen dieses Prinzip in unseren Übungen, indem wir Objekte untergeordneter Fenster in den Datenfeldern ihrer übergeordneten Fensterobjekte speichern. Diese Vorgehensweise ist zwar nicht unbedingt erforderlich, sie liefert jedoch klare, leicht zu wartende Programme.

Schritt 8: Hinzufügen eines Pop-up-Fensters

Als nächstes erstellen wir ein Pop-up-Fenster, das nach der Auswahl des Menüpunktes *Help* angezeigt wird, und das die Basis eines Hilfesystems für *MyProgram* bildet. Das Pop-up-Fenster soll eine Objektinstanz von *TWindow* werden.

In Schritt 10 werden wir einen neuen Typ für das Fenster erzeugen und ihm eine Verhaltensweise zuordnen. Momentan beschränken wir uns auf seine Anzeige auf dem Bildschirm.

Da es sich bei dem Pop-up-Fenster um das erste neue Fenster handelt, das wir hinzufügen (das Hauptfenster wird ja automatisch angezeigt), müssen wir uns zuerst mit der Erstellung und Anzeige von Fensterobjekten und -elementen befassen.

Erzeugen und Anzeigen des Pop-up-Fensters

In Schritt 7 haben wir auf die Auswahl des Menüpunkts *Help* durch die Anzeige eines Meldungsfensters reagiert. Dieses Meldungsfenster ersetzen wir jetzt durch ein leeres Pop-up-Fensterobjekt mit dem Namen *HelpWnd*.

In Schritt 10 werden wir die Verhaltensweisen für dieses leere Pop-up-Fenster definieren.

Momentan beschränken wir uns auf das Umdefinieren von *Help*: Das Hilfefenster soll erzeugt und angezeigt werden.

```
procedure TMyWindow.Help(var Msg: TMessage);
var HelpWnd: PWindow;
begin
  HelpWnd := New(PWindow, Init(@Self, 'Help System'));
  with HelpWnd^.Attr do
  begin
    Style := Style or ws_Visible or ws_PopupWindow or ws_Caption;
    X := 100;
    Y := 100;
    W := 300;
    H := 300;
  end;
  Application^.MakeWindow(HelpWnd);
end;
```

Abbildung 5.2 zeigt das neue Hilfefenster, das nach dem Anklicken von *Help* erscheint.

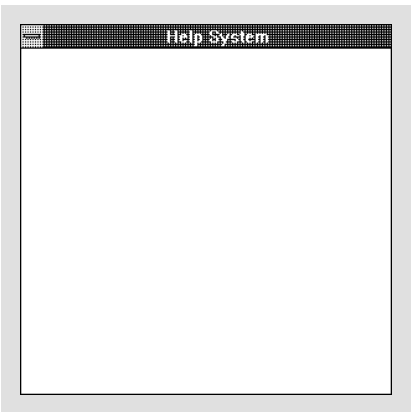


Abbildung 5.2 Das neue Hilfefenster von MyProgram

Die Funktion MakeWindow

Das Objekt *TApplication* definiert eine wichtige Methode namens *MakeWindow*. *MakeWindow* führt alle Aktionen aus, die für die sichere Verknüpfung eines Schnittstellenelements mit einem Fensterobjekt erforderlich sind. »Sicher« bedeutet in diesem Fall, daß *MakeWindow* eine Prüfung auf Fehlerbedingungen durchführt, *bevor* es zu ernsthaften Problemen (beispielsweise einem Systemabsturz) kommen kann. Die *MakeWindow*-Methode führt zwei Schritte aus:

1. Sie ruft *ValidWindow* auf, um das Schnittstellenobjekt auf seine Gültigkeit zu testen. *ValidWindow* überprüft, ob die Anwendung genügend Speicher zur Verfügung hat, um die Konstruktion des Schnittstellenobjekts abzuschließen.

2. Nach der erfolgreichen Konstruktion des Schnittstellenobjekts erstellt *MakeWindow* mit der *Create*-Methode des Objekts ein Schnittstellenelement. Über der Information in den Objektfeldern teilt *Create* Windows mit, welches Schnittstellenelement benötigt wird. Kann das Element nicht erstellt werden, wird eine Fehlermeldung angezeigt.

Kann die Gültigkeitsprüfung oder die Erstellung des Schnittstellenelements nicht korrekt durchgeführt werden, liefert *MakeWindow* nil zurück. Andernfalls liefert *MakeWindow* einen Zeiger auf das Schnittstellenelement zurück, das ihm als Parameter übergeben wurde.

Ein Aufruf von *MakeWindow* ist die sicherste Möglichkeit für die Erstellung von Schnittstellenelementen. Der direkte Aufruf von *Create* ohne die Überprüfung der Speicherkapazität und das Abfangen von Fehlern kann leicht zu einem Systemabsturz führen und ist deshalb nicht ratsam.

Hinzufügen eines Dialogfensters

Ein Dialogfenster ähnelt einem Pop-up-Fenster, bleibt aber normalerweise für kurze Zeit auf dem Bildschirm und führt eine – von einer Benutzereingabe abhängige – Aufgabe aus, beispielsweise die Auswahl eines Druckers. Das neue Dialogfenster für *MyProgram* soll das Öffnen und Speichern von Dateien ermöglichen.

Wie ein Pop-up-Fenster ist auch ein Dialogfenster ein selbständiges untergeordnetes Fenster. Im Prinzip entspricht die Konstruktion eines Dialogfensters als untergeordnetes Fenster dem Hinzufügen eines Pop-up-Fensters, das wir im letzten Abschnitt beschrieben haben. Ein Dialogfenster weist aber wichtige Unterschiede zu anderen Fenstern auf:

- Objekttypen für Dialogfenster sind nicht vom Typ *TWindow*, sondern vom Typ *TDialog* abgeleitet. Sowohl *TWindow* als auch *TDialog* sind von *TWindowsObject* abgeleitet.

- Dialogfenster benötigen normalerweise *Ressourcen*, die ihre Größe, Position und ihr Aussehen festlegen.

- Normalerweise führen Dialogfenster eine kurze Aufgabe aus und liefern einen Wert zurück. Beispielsweise liefert das *CanClose*-Meldungsfenster mit dem in Schritt 2 erstellten minimalen Dialog eine Yes- oder No-Antwort des Anwenders.

Unser nächster Schritt besteht im Hinzufügen des in ObjectWindows vordefinierten Datei-Dialogfensters. Es wird durch den von *TDialog* abgeleiteten Typ *TFileDialog* definiert. Das Datei-Dialogfenster kann in jeder Situation eingesetzt werden, in der der Anwender eine Datei zum Laden oder Speichern auswählen soll. In einem Textverarbeitungsprogramm dient ein Datei-Dialogfenster beispielsweise zum Öffnen und Speichern von Dokumenten.

Im unserem Beispiel soll ein Datei-Dialogfenster als Reaktion auf die Auswahl von *File/Open* oder *File/Save* angezeigt werden. Das Datei-Dialogfenster ersetzt somit das Meldungsfenster mit der Anzeige »Feature not implemented«. In

Schritt 9 werden wir es mit einigen Dateien verknüpfen, die geladen und gespeichert werden können. Jetzt wollen wir uns nur mit der Anzeige des Dialogfensters befassen. Abbildung 5.3 zeigt, wie unser Datei-Dialogfenster aussieht.

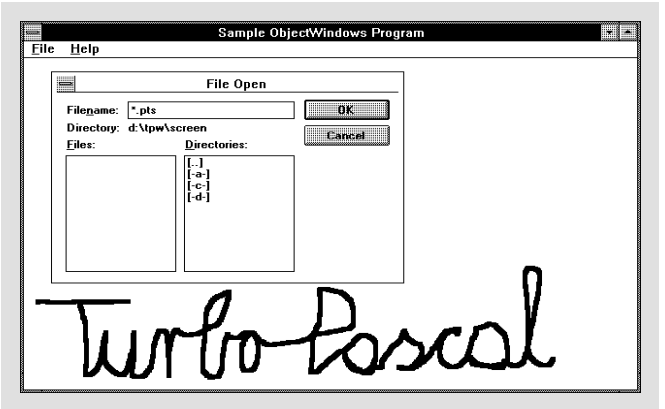


Abbildung 5.3 MyProgramm mit dem Datei-Dialogfenster

Hinzufügen eines Objektfeldes

Das Objekt eines Datei-Dialogfensters wird nicht wie beim Hilfefenster als Objektfeld seines übergeordneten Fensters gespeichert. Stattdessen wird für jedes benötigte Datei-Dialogfenster ein neues Objekt konstruiert. Gespeichert werden nur die Daten, die mit dem Datei-Dialogfenster verwendet werden sollen: ein Dateiname und eine Dateinamenmaske. Das hat den Vorteil, daß keine Objekte mitzuführen sind, die sonst nirgends benötigt werden. Es werden nur die Daten gespeichert, die für die Initialisierung der benötigten Objekte nötig sind.

Für die Konstruktion eines Datei-Dialogfensters werden drei Parameter benötigt: ein übergeordnetes Fenster, eine Ressourcenvorlage und ein Dateiname bzw. eine Dateinamenmaske (je nachdem, ob das Fenster zum Öffnen oder Schließen einer Datei verwendet werden soll). Die Ressourcenvorlage legt fest, welches Standard-Datei-Dialogfenster benutzt werden soll. Die Standard-Datei-Dialogressourcen werden durch die Ressourcen-IDs *sd_FileOpen* und *sd_FileSave* bezeichnet. Über den Dateinamensparameter wird an das Dialogfenster zum Öffnen von Dateien eine voreingestellte Dateinamenmaske übergeben (die Rückgabe des gewählten Dateinamens erfolgt ebenfalls über diesen Parameter). Entsprechend erhält auch das Dialogfenster zum Speichern von Dateien den voreingestellten Namen über den Dateinamensparameter.

Mit dem Parameter für die Ressourcenvorlage wird festgelegt, ob das Dialogfenster zum Öffnen oder zum Speichern von Dateien verwendet werden soll. Hat die Dialogressource eine Dateiliste mit der Steuerelement-ID *id_FList*, dient das Fenster zum Öffnen von Dateien. Das Fehlen dieser Liste zeigt an, daß das Dialogfenster zum Speichern verwendet wird.

Die Typdefinition von *TMyWindow* sieht nun folgendermaßen aus:

```
TMyWindow = object(TWindow)
  ...
  FileName: array[0..fsPathName] of Char;
  ...
```

Ändern des Init-Konstruktors

Zur Instantiierung des Hilfefensterobjekts wurde bereits ein *Init*-Konstruktor für *TMyWindow* angelegt. Zu diesem wird nun der Code für die Initialisierung von *FileName* hinzugefügt:

```
StrCopy(FileName, '*.PTS');
```

Die Namenserweiterung .PTS wird für die Dateien benutzt, die die Punkte der Zeichnungen enthalten.

Anzeigen des Dialogfensters

Abhängig von dem Parameter für die Ressourcenvorlage, der an den Konstruktor des Datei-Dialogobjekts übergeben wird, kann das Dialogfenster entweder zum Öffnen oder zum Speichern von Dateien eingesetzt werden. In beiden Fällen wird ein Dialogfenster ähnlich dem in Abbildung 5.3 gezeigten Fenster erzeugt.

Dialogfenster zum Öffnen von Dateien und zum Speichern von Dateien unterscheiden sich in zwei Punkten: Das Dialogfenster zum Öffnen enthält eine Liste mit den Dateien im aktuellen Verzeichnis, die der angegebenen Dateinamenmaske entsprechen. Dagegen zeigt das Dialogfenster zum Speichern im Editierbereich seines Editierfeldes nur den aktuellen Dateinamen an.

FileOpen und *FileSaveAs* werden folgendermaßen umgeschrieben:

```
procedure TMyWindow.FileOpen(var Msg: TMessage);
begin
  if Application^.ExecDialog(New(PFileDialog,
    Init(@Self, PChar(sd_FileOpen, StrCopy(FileName, '*.PTS'))))
    = id_Ok then
    MessageBox(HWindow, FileName, 'Open the file:', mb_OK);
end;

procedure TMyWindow.FileSaveAs(var Msg: TMessage);
begin
  if Application^.ExecDialog(New(PFileDialog,
    Init(@Self, PChar(sd_FileSave), FileName))) = id_OK then
    MessageBox(HWindow, FileName, 'Save the file:', mb_OK);
end;
```

Der aktuelle Stand des Quelltextes von *MyProgram* ist in STEP08.PAS enthalten.

Wir erinnern uns: Zur »sicheren« Erstellung von Schnittstellenelementen haben wir *TApplication.MakeWindow* verwendet. Auf ähnliche Weise lassen sich mit *TApplication.ExecDialog* sichere Dialogfenster erzeugen und anzeigen. Wie *MakeWindow* überprüft auch *ExecDialog* mit Hilfe von *ValidWindow*, ob die Konstruktion des Dialogobjekts erfolgreich war. Ist das der Fall, ruft *ExecDialog* die *Execute*-Methode des Dialogobjekts auf. Nach einer fehlerlosen Ausführung liefert *ExecDialog* den von *Execute* zurückgegebenen Wert zurück. Bei einem Fehler wird *id_Cancel* zurückgeliefert. Für Ihr Programm sieht das aus wie die Auswahl von Cancel durch den Anwender. In beiden Fällen gibt *ExecDialog* das Objekt des Dialogfensters frei.

Ein Aufruf von *ExecDialog* ist die sicherste Möglichkeit zur Durchführung eines Dialogs. Der direkte Aufruf von *Execute* wäre zwar auch möglich, könnte aber zum Systemabsturz führen und ist deshalb nicht zu empfehlen.

Schritt 9: Speichern der Zeichnung in einer Datei

Die Zeichnung liegt nun als Teil des Fensterobjekts in einer Form vor, die als Datei (genauer gesagt als DOS-Stream) gespeichert und wieder eingelesen werden kann. Zum Speichern des Status und zum Modifizieren der Dateispeicherungs- und Dateiöffnungsmethoden benötigen wir noch einige Objektfelder, die wir als nächstes definieren werden.

Überwachung des Status

Zwei Eigenschaften der Zeichnung müssen im Auge behalten werden: »Muß die Datei gespeichert werden?« und »Ist momentan eine Datei geladen?«. Diese Fragen können in Boolesche Attribute von *TMyWindow* umgesetzt und als Objektfelder dargestellt werden:

```
TMyWindow = object(TWindow)
  ...
  IsDirty, IsNewFile: Boolean;
  ...
end;
```

IsDirty ist *True*, wenn die Datei gespeichert werden muß, da sie seit der letzten Änderung modifiziert wurde oder bisher noch gar nicht gespeichert worden ist. *IsDirty* muß auf *True* gesetzt werden, wenn der Anwender zu zeichnen beginnt (*WMLButtonDown*). Öffnet der Anwender eine neue Datei oder speichert er die vorhandene, wird *IsDirty* auf *False* gesetzt. Beendet er das Programm (*CanClose*), sollte der Status von *IsDirty* überprüft und, falls er *True* ist, ein Meldungsfenster angezeigt werden.

IsNewFile ist nur dann *True*, wenn das Programm das erste Mal gestartet wird (*Init*) und wenn der Anwender das Menü *File/New* wählt (*FileNew*). Durch das Öffnen (*FileOpen*) oder das Schließen (*FileSave* oder *FileSaveAs*) einer Datei wird *IsNewFile* auf *False* gesetzt. *FileSave* stellt mit Hilfe von *IsNewFile* fest, ob die Datei sofort gespeichert werden kann (*False*) oder ob der Anwender eine Datei aus dem Datei-Dialogfenster wählen muß (*True*).

Das folgende Programmlisting enthält die *CanClose*-Methode und die Methoden für das Speichern und Laden von Dateien. Das tatsächliche Speichern bzw. das Öffnen der Datei wird von den Methoden *SaveFile* und *LoadFile* übernommen.

```
function TMyWindow.CanClose: Boolean;
var
  Reply: Integer;
begin
  CanClose := True;
  if IsDirty then
  begin
    Reply := MessageBox(HWindow, 'Do you want to save?',
      'Drawing has changed', mb_YesNo or mb_IconQuestion);
    if Reply = id_Yes then CanClose := False
  end;
end;

procedure TMyWindow.FileNew(var Msg: TMessage);
  begin
    Points^.FreeAll;
    InvalidateRect(HWindow; nil, True);
    IsDirty := False;
    IsNewFile := True;
end;

procedure TMyWindow.FileOpen(var Msg: TMessage);
begin
  if CanClose then
    if Application^.ExecDialog(New(PFileDialog, Init(@Self,
      PChar(sd_FileOpen), StrCopy(FileName, '*.PTS')))) = id_OK then
      LoadFile;
end;

procedure TMyWindow.FileSave(var Msg: TMessage);
begin
  if IsNewFile then FileSaveAs(Msg) else SaveFile;
end;

procedure TMyWindow.FileSaveAs(var Msg: TMessage);
var
  FileDlg: PFileDialog;

begin
  if IsNewFile then StrCopy(FileName, '');
```

```
  if Application^.ExecDialog(New(PFileDialog,
      Init(@Self, PChar(sd_FileSave), FileName))) = id_Ok then
    SaveFile;
end;

procedure TMyWindow.LoadFile;
begin
  MessageBox(HWindow, @FileName, 'Load the file:', mb_OK);
  IsDirty := False;
  IsNewFile := False;
end;

procedure TMyWindow.SaveFile;
begin
  MessageBox(HWindow, @FileName, 'Save the file:', mb_OK);
  IsNewFile := False;
  IsDirty := False;
end;
```

Speichern und Laden von Dateien

Nachdem wir das Gerüst zum Speichern und Laden von Dateien erstellt haben, können wir jetzt die Zeichnung (d.h. die Kollektion von Punkten) in einer Datei speichern. Dazu verwenden wir den Mechanismus zum Speichern von Objekten in Streams. Zunächst müssen die Punkte in die Lage versetzt werden, sich selbst zu speichern und zu laden (Kollektionen können dies bereits). Anschließend werden die Methoden *SaveFile* und *FileOpen* modifiziert, um ihnen die Benutzung von Streams zu ermöglichen.

Im folgenden sehen Sie den Code, der es den *TDPoint*-Objekten ermöglicht, sich selbst zu speichern und zu laden:

```
type
  PDPoint = ^TDPoint;
  TDPoint = object(TObject)
    X, Y: Integer;
    constructor Init(AX, AY: Integer);
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

const
  RDPoint: TStreamRec = (
    ObjType: 200;
    VmtLink: Ofs(TypeOf(TDPoint)^);

    Load: @TDPoint.Load;
    Store: @TDPoint.Store);

procedure StreamRegistration;
```

```
begin
  RegisterType(RCollection);
  RegisterType(RDPoint);
end;

constructor TDPoint.Load(var S: TStream);
begin
  S.Read(X, SizeOf(X));
  S.Read(Y, SizeOf(Y));
end;

procedure TDPoint.Store(var S: TStream);
begin
  S.Write(X, SizeOf(X));
  S.Write(Y, SizeOf(Y));
end;
```

Beim Start der Anwendung muß *StreamRegistration* aufgerufen werden. Dieser Aufruf kann zur *TMyWindow.Init*-Methode hinzugefügt werden.

Der letzte Schritt besteht im Neuschreiben der Methoden *SaveFile* und *FileOpen*:

```
procedure TMyWindow.LoadFile;
var
  TempColl: PCollection;
  TheFile: TDosStream;
begin
  TheFile.Init(FileName, stOpen);
  TempColl := PCollection(TheFile.Get);
  TheFile.Done;
  if TempColl <> nil then
  begin
    Dispose(Points, Done);
    Points := TempColl;
    InvalidateRect(HWindow, nil, True);
  end;
  IsDirty := False;
  IsNewFile := False;
end;

procedure TMyWindow.SaveFile;
var
  TheFile: TDosStream;
begin
  TheFile.Init(FileName, stCreate);
  TheFile.Put(Points);
  TheFile.Done;
  IsNewFile := False;
  IsDirty := False;
end;
```

6. Anzeigen von Fenstern

Zum Schluß wollen wir noch ein Hilfesystem für unser ObjectWindows-Programm implementieren. Sie können sich das Hilfesystem als ein Programm im Programm vorstellen, das jederzeit aktiviert werden kann. Windows-Programme haben hier aufgrund ihrer fensterorientierten Umgebung den Vorteil, daß viele Aktivitäten zusammengefaßt werden können. Anhand der Erstellung des Hilfesystems für *MyProgram* soll Ihnen vor Augen geführt werden, wie man ObjectWindows-Programme durch den Einsatz von Units in einem modularen Format entwickelt und dadurch die Programmpflege erleichtert.

Schritt 10: Anzeigen eines Hilfefensters

In Schritt 8 haben wir zum Hauptfenster von *MyProgram* ein Hilfemenü hinzugefügt. Die Auswahl des Menüpunktes *Help* hat jedoch nur die Anzeige eines leeren Fensters zur Folge, das eine Objektinstanz von *TWindow* ist. Wir werden nun dieses leere Fenster durch ein sinnvolles Hilfefenster ersetzen und dabei einige Windows-Steuerelemente von ObjectWindows verwenden: Listen, Bildlaufleisten, Editierfelder, statische Steuerelemente, Schalter und Kombinationsfenster. Das Hilfesystem wird als Unit definiert, wobei nur geringe Änderungen im momentanen Quelltext erforderlich sind. Abbildung 6.1 zeigt das fertige Hilfesystem, das für jedes Steuerelement von ObjectWindows eine kurze Beschreibung liefert.

Die Verwendung von Units mit ObjectWindows

Eine Turbo-Pascal-Unit bietet den idealen Rahmen zur Definition von Objekten oder Gruppen miteinander verknüpfter Objekte. Dies ist vor allem bei der Programmierung von Windows-Anwendungen sinnvoll, da hier oft dieselben Fenster (Fensterobjekte in ObjectWindows) von verschiedenen Anwendungen benutzt werden. Wir konstruieren unser Hilfesystem als Objekttyp mit dem Namen *THelpWindow* und speichern seine Definition in der Unit *HelpWind*.

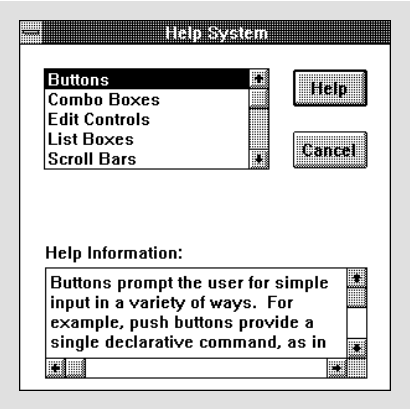


Abbildung 6.1
Das Hilfesystem von MyProgram

Ändern des Hauptprogramms

Es sind nur zwei Änderungen an *MyProgram* nötig:

- Hinzufügen der Unit *HelpWind* in der uses-Anweisung.
- Entfernen der Programmteile aus TMyWindow.Help, die Attribute von HelpWind erzeugen. Der betreffende Code ist nun in THelpWindow.Init enthalten.

MyProgram reagiert jetzt auf die Auswahl des Menüpunktes *Help* durch die Anzeige eines Hilfefensters (*THelpWindow*). Wir müssen daher noch den *THelpWindow*-Typ definieren. Dies erfolgt in der Unit *HelpWind*.

Erzeugen der Unit

Die Unit *HelpWind* dient ausschließlich zur Definition des Interface- und des Implementationsteils des Typs *THelpWindow*. Nach ihrer Erstellung kann diese Unit von jedem anderen ObjectWindows-Programm genutzt werden. Sie ist im Grunde nichts anderes als ein Fenster in einer Unit. Das Format der Unit sieht folgendermaßen aus (die Lücken werden wir später noch füllen):

```
unit HelpWind;
Interface
uses Strings, WObjects, WinTypes, WinProcs;

const
  id_LB1 = 201;
  id_BN1 = 202;
  id_BN2 = 203;
  id_EC1 = 204;
  id_ST1 = 205;
```

```
type
  PHelpWindow = ^THelpWindow;
  THelpWindow = object(TWindow)
    LB1: PListBox;
    EC1: PEdit;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure SetupWindow; virtual;
    procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
    procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
    procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
    procedure FillEdit(SelStringPtr: PChar); virtual;
end;

Implementation

constructor THelpWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  ...
end;

procedure THelpWindow.SetupWindow;
begin
  ...
end;

procedure THelpWindow.IDLB1(var Msg: TMessage);
var
  ...
begin
  ...
end;

procedure THelpWindow.IDBN1(var Msg: TMessage);
var
  ...
begin
  ...
end;

procedure THelpWindow.IDBN2(var Msg: TMessage);
begin
  ...
end;

procedure THelpWindow.FillEdit(SelStringPtr: PChar);
var
  ...
begin
  ...
end;

end.
```

Ist *HelpWind* in der uses-Anweisung einer ObjectWindows-Anwendung enthalten, kann diese den Typ *THelpWindow* durch die Definition von Variablen als *PHelpWindow*-Typ instantiieren.

Steuerelemente in einem Fenster

Aus Schritt 8 wissen wir, daß es zwei Arten von untergeordneten Fenstern gibt: selbständige und unselbständige. In dem betreffenden Abschnitt wurden zwei Typen von selbständigen untergeordneten Fenstern zum Hauptfenster hinzugefügt: Dialogfenster und Pop-up-Fenster.

Jetzt wollen wir ein unselbständiges untergeordnetes Fenster hinzufügen, ein *Steuerelement*. Die Steuerelemente sind dem Hilfefenster direkt untergeordnet. Unser Hilfefenster ist ein selbständiges, dem Hauptfenster der Anwendung untergeordnetes Fenster. Es ist daher gleichzeitig ein übergeordnetes und ein untergeordnetes Fenster.

Dieser Status ist zulässig und in fast allen komplexeren Windows-Anwendungen auch erforderlich.

Was sind Steuerelemente?

Steuerelemte sind sichtbare Einheiten, aus denen sich die Benutzerschnittstelle des Fensters (oder ein Teil davon) zusammensetzt. Beispiele für Steuerelemente sind die Schalter *Save* und *Open* in Datei-Dialogfenstern oder die Kombinationsfenster zur Anzeige von Dateien. Das Hauptfenster von *MyProgram* enthält keine Steuerelemente, wir wollen aber einige in unser Hilfefenster aufnehmen.

Es gibt verschiedene Arten von Steuerelementen wie Listen, Bildlaufleisten, Editierfelder, statische Steuerelemente, Schalter und Kombinationsfenster. Unser Hilfefenster (siehe Abbildung 6.1) enthält eine Liste (das große Fenster in der oberen Hälfte), ein Editierfeld (das große Fenster in der unteren Hälfte), ein statisches Steuerelement (der Text »Help Information:«) und zwei Schalter (Help und Cancel).

Bildlaufleisten sind im Editierfeld und in der Liste enthalten, es handelt sich aber dabei nicht um Bildlaufleistenobjekte. Sie werden meist als optionaler Bestandteil von anderen Steuerelementen und Fenstern eingesetzt, dagegen nur selten als selbständiges Steuerelement verwendet.

Der ObjectWindows-Typ *TControl* enthält die allgemeinen Verhaltensweisen und Eigenschaften aller Steuerelemente, die von *TControl* abgeleiteten Typen definieren die verschiedenen Arten von Steuerelementen. Zum Beispiel definiert *TListBox* Listenobjekte und *TEdit* Editierfeldobjekte. *TControl* ist von *TWindows* abgeleitet. Im Grunde sind Steuerelemente nichts anderes als »spezialisierte« Fenster.

Erstellen von Steuerelementen

Steuerelemente in Dialogen (zum Beispiel Datei-Dialogfenster) sind zwar in ihrer Verhaltensweise identisch mit Steuerelementen in Fenstern (zum Beispiel in unserem Hilfefenster), bei ihrer Programmierung gibt es aber einen wichtigen Unterschied. Steuerelemente eines Dialogfensters werden in der Dialogressource festgelegt. Sie sind keine Objekte und die Dialogfenster, zu denen sie gehören, sind allein für ihre Verwaltung zuständig.

Kapitel 5 »Konstruktion eines Dialogs« erläutert, wie aus den Dialogressourcen eigene Dialogfenster erstellt und wie ihre Steuerelemente definiert werden können.

Die Steuerelemente eines Fensters werden durch Objektdefinitionen festgelegt. Ein übergeordnetes Fenster verwaltet seine Steuerelemente über die vielen Methoden, die durch die Steuerelementobjekte von ObjectWindows definiert sind. Zum Beispiel wird zur Übernahme des Textes, den ein Anwender aus einer Liste gewählt hat, die Methode *GetSelString* des Listenobjekts aufgerufen. Ähnlich wie bei einem Fenster- oder Dialogfensterobjekt existiert auch zu jedem Steuerelementobjekt ein entsprechendes sichtbares Element. Ein Steuerelementobjekt ist mit seinem zugehörigen sichtbaren Element durch das ID-Feld verknüpft. Jedes Steuerelement hat eine eindeutige ID-Nummer, durch die es von seinem übergeordneten Fenster beim Auftreten eines Steuerelement-Ergebnisses (zum Beispiel ein Mausklick) identifiziert werden kann. Um das Programm übersichtlicher zu machen, definieren wir für jede Steuerelement-ID eine Konstante:

```
const
id_LB1 = 201;          { ID für die Liste 1 }
id_BN1 = 202;          { ID für den ersten Schalter (OK) }
id_BN2 = 203;          { ID für den zweiten Schalter (Cancel) }
id_EC1 = 204;          { ID für das Editierfeld }
id_ST1 = 205;          { ID für statischen Text }
```

Hinweis: Dialog-Steuerelemente können auch mit Objekten verknüpft sein (vergleichen Sie den Abschnitt »Verknüpfen von Steuerelementobjekten« in Kapitel 11 »Dialogobjekte«).

Steuerelementobjekte als Felder

Manchmal ist es sinnvoll, einen Zeiger auf ein Steurelementobjekt (oder auf ein anderes untergeordnetes Fenster) als Feld in einem Fensterobjekt zu speichern. Es ist dann später möglich, die entsprechenden untergeordneten Fenster direkt durch einen Aufruf ihrer Objektmethoden zu manipulieren. In unserem Fall trifft das nur für die Liste und das Editierfeld zu. *THelpWindow* speichert jedes dieser Steuerelementobjekte in einem separaten Feld.

In dem folgenden Listing sehen Sie einen Teil der Objektdefinition von *THelpWindow*:

```
THelpWindow = object(TWindow)
  LB1: PListBox;
  EC1: PEdit;
  ...
end;
```

Nach der Instantiierung dieser Objekte können die Steuerelemente mit Methodenaufrufen geändert werden. Beispielsweise könnte durch den Aufruf von *LB1^.AddString* ein neuer String in die Liste aufgenommen werden. Es wäre zwar auch möglich, über die *ChildList* des übergeordneten Fensters auf Steuerelementobjekte von untergeordneten Fensters zuzugreifen, die nicht als Felder gespeichert sind, die Verwendung von Feldern ist aber bedeutend einfacher.

Verwalten von Steuerelementen

Jeder Fenstertyp, der über Steuerelementobjekte verfügt, muß zur Konstruktion dieser Objekte einen Konstruktor (*Init*) definieren. Soll das Einrichten der Steuerelemente vor ihrer Anzeige erfolgen, kann der Fenstertyp außerdem *SetupWindow* überschreiben. Das übergeordnete Fenster (*THelpWindow*) erzeugt seine sämtlichen untergeordneten Fenster und zeigt diese an.

Das nächste Programmlisting enthält die *Init*-Methode des Hilfefensters. Zunächst werden die Positions- und Größenattribute gesetzt. Da in der *Init*-Methode die Erstellungsattribute des Fensters gesetzt werden, müssen hier auch die Steuerelemente erzeugt werden. Im Aufruf jedes Steuerelement-Konstruktors ist der erste Parameter das übergeordnete Fenster (*@Self*), der zweite die Steuerelement-ID.

```
constructor THelpWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
  TempStat: PStatic;
  TempBtn: PButton;
begin
  TWindow.Init(AParent, ATitle);
  DisableAutoCreate;
  Attr.Style := ws_PopupWindow or ws_Caption or ws_Visible;
  Attr.X := 100;
  Attr.Y := 100;
  Attr.W := 300;
  Attr.H := 300;
  LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 180, 80));
  TempBtn := New(PButton, Init(@Self, id_BN1, 'Help', 220, 20,
    60, 30, True));
  TempBtn := New(PButton, Init(@Self, id_BN2, 'Cancel', 220, 70, 60,
    30, False));
  EC1 := New(PEdit, Init(@Self, id_EC1, '', 20, 180, 260, 90, 40,
   True));
  EC1^.Attr.Style := EC1^.Attr.Style or ws_Border or ws_VScroll;
```

```
    TempStat := New(PStatic, Init(@Self, id_ST1, 'Help Information:',
      20, 160, 160, 20));
end;
```

Für die Einrichtung der Steuerelemente des Fensters wird nach der Erzeugung des Fensters *THelpWindow.SetupWindow* aufgerufen. Das Füllen der Liste mit Einträgen erfolgt mit Hilfe der Modifizierungsmethoden von *TListBox* (zum Beispiel *AddString*).

Wird die *SetupWindow*-Methode überschrieben, muß dort als erstes zur Erzeugung der untergeordneten Steuerelemente *TWindow.SetupWindow* aufgerufen werden.

```
procedure THelpWindow.SetupWindow;
begin
  TWindow.SetupWindow;
  { Füllen der Liste }
  LB1^.AddString('List Boxes');
  LB1^.AddString('Buttons');
  LB1^.AddString('Scroll Bars');
  LB1^.AddString('Edit Controls');
  LB1^.AddString('Static Controls');
  LB1^.AddString('Combo Boxes');
  LB1^.SetSelIndex(0);
end;
```

Die Methoden *Init* und *SetupWindow* reichen aus, um alle Steuerelemente korrekt im Hilfefenster anzuzeigen. Die Liste kann gerollt und die Schalter können aktivert werden, es wird aber keine Aktion ausgeführt.

Im nächsten Abschnitt definieren wir eine Antwort auf Steuerelement-Ereignisse.

Antworten auf Steuerelement-Ereignisse

Die Steuerelemente werden jetzt zwar im Hilfefenster angezeigt, das Anklicken der Schalter und die Auswahl aus der Liste hat jedoch keine Aktion zur Folge. Beim Anklicken und Auswählen handelt es sich um Steuerelement-Ereignisse.

Steuerelement-Ereignisse ähneln den Menüereignissen, für die wir in Schritt 7 Methoden zur Antwort auf Befehlsbotschaften definiert haben. Bei Steuerelement-Ereignissen wird auf ähnliche Weise verfahren. Allerdings erzeugen Steuerelement-Ereignisse Botschaften, die auf der ID des untergeordneten Fensters basieren und deshalb die Steuerelement-ID anstelle der Menü-ID übergeben.

Zur Identifizierung eines Methodenkopfes, der auf der ID des untergeordneten Fensters basiert, verwenden wir die Summe aus der Steuerelement-ID und der Konstanten *id_First*. Um die Lesbarkeit zu verbessern, erhalten diese Botschaftsantwort-Methoden Namen wie *IDLB1*.

Die folgende *THelpWindow*-Typdefinition zeigt die Methodenköpfe:

```
THelpWindow = object(TWindow)
  LB1: PListBox;
  EC1: PEdit;
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure SetupWindow; virtual;
  procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
  procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
  procedure FillEdit(SelStringPtr: PChar); virtual;
end;
```

Als nächstes wird festgelegt, wie *THelpWindow* auf Steuerelement-Ereignisse reagiert. Klickt der Anwender den Schalter Cancel (BN2) an, soll das Fenster geschlossen werden. Die Implementierung von *IDBN2* ist daher sehr einfach:

```
procedure THelpWindow.IDBN2(var Msg: TMessage);
begin
  CloseWindow;
end;
```

Klickt der Anwender dagegen den Schalter OK (BN1) an, soll *MyProgram* überprüfen, welcher Punkt in der Liste gewählt wurde, und das Editierfeld (EC1) mit dem entsprechenden Text füllen. *IDBN1* ruft deshalb *GetSelString* für die Liste (LB1) auf, um den Text für den gewählten Listenpunkt zu erhalten. Der Text wird an die Methode *FillEdit* übergeben. Die Definition von *FillEdit* ist in dem vollständigen Programmlisting in STEP10.PAS enthalten.

```
procedure THelpWindow.IDBN1(var Msg: TMessage);
var
  SelString: array[0..25] of Char;
begin
  LB1^.GetSelString(@SelString, 25);
  FillEdit(@SelString);
end;
```

Zum Abschluß wollen wir noch auf die Botschaft antworten, die durch das doppelte Anklicken eines Listenpunktes erzeugt wird. Das Editierfeld soll in diesem Fall ebenfalls mit dem entsprechenden Text gefüllt werden. Ein Liste kann nun aber als Reaktion auf einen Mausklick, einen Doppelklick, Rollen etc. verschiedene Botschaften erzeugen (sogenannte Listen-Benachrichtigungsbotschaften).

Damit zwischen den Botschaften unterschieden werden kann, muß *IDLB1* den Parameter *Msg.ParamHi* prüfen, in dem die Listenbenachrichtigung enthalten ist. Ist der Wert *lbn_DblClk*, hat der Anwender doppelt angeklickt. Wir ignorieren hier alle anderen Werte, die von der Standard-Fensterprozedur *DefWndProc* bearbeitet werden.

```
procedure THelpWindow.IDLB1(var Msg: TMessage)
var
  SelString: array[0..25] of Char;
begin
```

```
    if Msg.LParamHi = lbn_DblClk then
    begin
      LB1^.GetSelString(SelString, 25);
      FillEdit(SelString);
    end
  end;
```

Mit den Steuerelementobjekten von ObjectWindows steht Ihnen eine Vielzahl von Abfrage- und Änderungsmethoden zur Verfügung, die Sie zur Gestaltung Ihrer Fenster einsetzen können.

II. Die Verwendung von ObjectWindows

7. Überblick über ObjectWindows

ObjectWindows, die Windows-Objektbibliothek, ist eine umfassende Zusammenstellung von Objekttypen, die die Entwicklung von Microsoft-Windows-Programmen mit Turbo Pascal auf effiziente Weise unterstützen.

In dem vorliegenden Kapitel finden Sie einen Überblick über die Objekthierarchie und deren Komponenten in ObjectWindows. In den restlichen Kapiteln in diesem Teil des Handbuchs werden die verschiedenen Hierarchie-Ebenen detailliert erläutert.

Zusätzlich zur Beschreibung der Objekthierarchie erfahren Sie in diesem Kapitel einiges über die grundlegenden Porgrammierprinzipien für die Windows-Umgebung, einschließlich der Verwendung von Ressourcen, des Aufrufs der API und des Empfangs und der Bearbeitung von Windows-Botschaften.

ObjectWindows-Konventionen

Die Namen der Objekttyen in ObjectWindows beginnen mit einem T. So erstellt der Typ *TDialog* Dialogfensterobjekte. Zu jeder Typdefinition gehört ein Zeiger auf den Typ. Der Name des Zeigers beginnt mit einem P. So lautet der Name des Zeigers auf *TDialog* beispielsweise *PDialog*. In allen Beispielen im vorliegenden Handbuch werden dynamische Objektinstanzen, wie zum Beispiel *PDialog*, und nicht *TDialog*-Objekte erstellt.

Botschaftsantwort-Methoden werden nach der zu beantwortenden Botschaft benannt, führen jedoch keinen Unterstrich in ihrem Namen. So würde eine Methode, die auf die Botschaft *wm_KeyDown* antwortet, *WMKeyDown* heißen und eine Anwortmethode auf die Botschaft *cm_FileOpen* hätte den Namen *CMFileOpen*.

Die ObjectWindows-Hierarchie

ObjectWindows ist eine Bibliothek, die aus einer Hierarchie von Objekttypen besteht, die Sie – unter Ausnützung der Vererbung – verwenden, ändern und

ergänzen können. Kapitel 5 »Verzeichnis der Objekte« im *Windows-Referenzhandbuch* enthält eine vollständige Referenz für diese Typen und ihre Felder und Methoden.

Das vorliegende Handbuch ist eine umfassende Anwendungsanleitung für die Objekttypen und Methoden der Bibliothek. Es enthält Beispielprogramme, die Sie schrittweise durch den Aufbau von Windows-Programmen mit Object-Windows führen.

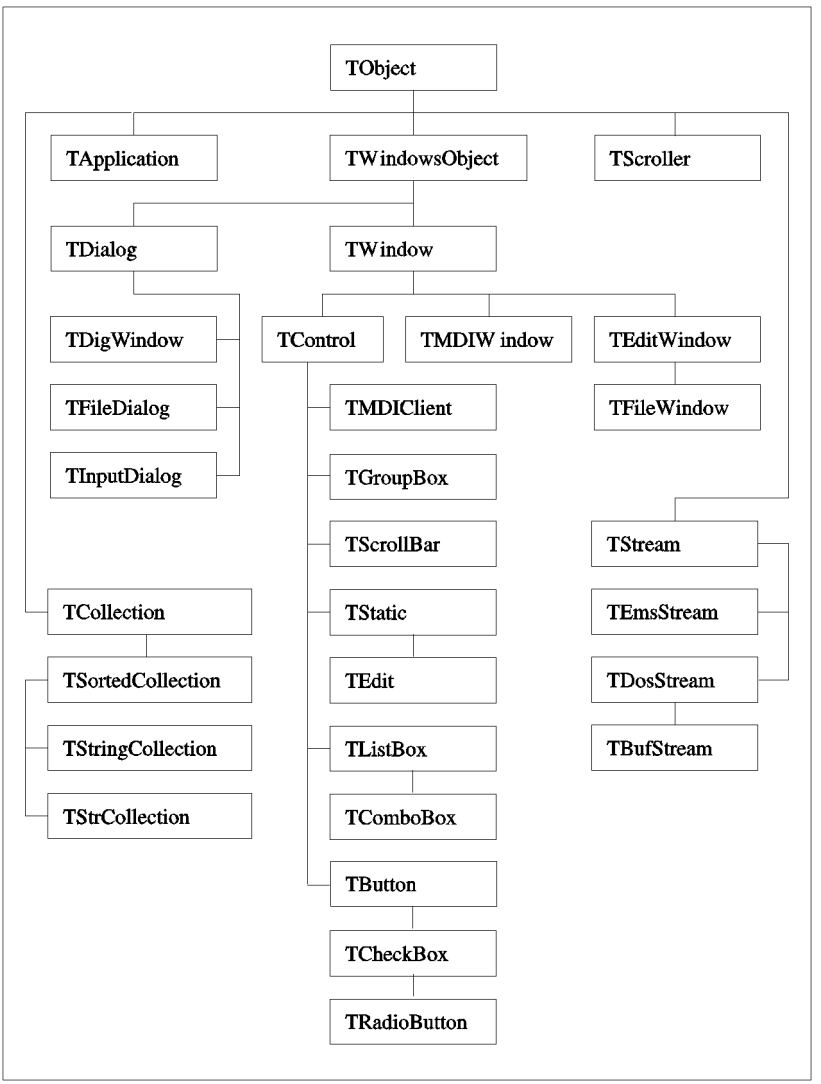


Abbildung 7.1 Die Hierarchie in ObjectWindows

Basis-objekte

TObject ist der Basisobjekttyp, das heißt der allen ObjectWindows-Objekttypen gemeinsame Vorfahre. Er definiert einen rudimentären Konstruktor und Destruktor. ObjectWindows-Streams erfordern, daß die in ihnen gespeicherten Objekte Nachkommen von *TObject* sind.

TApplication

Dieser Typ bestimmt das Verhalten und die Eigenschaften für alle ObjectWindows-Anwendungen. Jede ObjectWindows-Anwendung, die Sie schreiben, definiert einen Anwendungstyp, der von *TApplication* abstammt. Er ist unter anderem für die Initialisierung des Hauptfensterobjekts verantwortlich. Anwendungsobjekte werden detailliert in Kapitel 8 »Anwendungsobjekte« erläutert.

Schnittstellen-objekte

Die restlichen Objekte in der ObjectWindows-Hierarchie werden generell als *Schnittstellenobjekte* klassifiziert. Sie sind dies in zweierlei Hinsicht: Sie repräsentieren Elemente in der Windows-Anwenderschnittstelle und dienen als eine Art Schnittstelle zwischen Ihrem Anwendungscode und der Windows-Umgebung. Schnittstellenobjekte werden in Kapitel 9 »Schnittstellenobjekte« detailliert erläutert.

TWindowsObject

TWindowsObject ist ein abstrakter Objekttyp, der die drei Haupttypen von ObjectWindows-Schnittstellenobjekten in sich vereint: Fenster, Dialogfenster und Steuerelemente. Dieser Typ stellt Methoden zur Verfügung, die die Erstellung, Botschaftsbearbeitung und die Freigabe von Fensterobjekten verwalten.

Fenster-objekte

Fensterobjekte sind nicht nur die bekannten Fenster der Windows-Umgebung, sondern auch die meisten der visuellen Elemente innerhalb der Umgebung, wie zum Beispiel Steuerelemente.

TWindow

TWindow-Objekte sind »Allzweck«-Fenster, die als Hauptfenster einer Anwendung oder als eines ihrer Pop-up-Fenster dienen können. Objektinstanzen von *TWindow* können Grafiken anzeigen. Sie werden aber in der Praxis abgeleitete Typen definieren, um das Verhalten und die Eigenschaften von *TWindow* spezieller zu gestalten.

TEditWindow
Ein Nachkomme von *TWindow*, *TEditWindow*, definiert ein Fenster, in dem Text editiert werden kann.

TFileWindow
Als Nachkomme von *TEditWindow* definiert *TFileWindow* ein Fenster, das Textdateien lädt, editiert und speichert.

Dialog-objekte

Dialogobjekte erleichtern den Umgang mit interaktiven Gruppen, vor allem mit Steuerelementgruppen wie Schaltern, Listen und Bildlaufleisten. Dialogobjekte werden in Kapitel 11 »Dialogobjekte« detailliert erläutert.

TDialog
Dieser abstrakte Objekttyp dient als Vorlage für abgeleitete Typen, die Windows-Dialogfenster verwalten. Dialogobjekte sind mit Dialogressourcen verknüpft und können als modale oder nicht-modale Dialogfenster eingesetzt werden.
Außerdem stehen Methoden zur Verfügung, die die Kommunikation zwischen einem Dialog und seinen Steuerelementen bewerkstelligen.

TDlgWindow
Dieser Typ kombiniert die Möglichkeit der Definition von Dialogobjekten in Ressourcen mit der individuellen Realisierung von Fensterobjekten.

TFileDialog
TFileDialog ist ein vordefinierter Dialogtyp, der in vielen Anwendungen sehr hilfreich ist. Er definiert einen Dialog, der dem Anwender die Auswahl einer Datei für verschiedene Zwecke ermöglicht, wie zum Beispiel zum Öffnen, Editieren oder Speichern.

TInputDialog
Dieser Typ definiert ein vordefiniertes Dialogfenster, das dem Anwender die Eingabe eines einzelnen Textelements erlaubt.

Steuerelement-objekte

In Dialog- und einigen anderen Fenstern erlauben Steuerelemente dem Anwender, Daten einzugeben und Optionen auszuwählen. Steuerelementobjekte stellen konsistente und einfache Mittel für den Umgang mit den verschiedenen, von Windows definierten Steuerelementen zur Verfügung.
Diese Objekte werden in Kapitel 12 »Steuerelementobjekte« detailliert erläutert.

TControl
TControl ist ein abstrakter Objekttyp, der als gemeinsamer Vorfahre für alle Windows-Steuerelemente dient, einschließlich Listen und Schalter. Er definiert Methoden, die für seine abgeleiteten Typen Steuerelemente anlegen und Botschaften bearbeiten.

TButton
TButton regelt das Erstellen von Windows-Aktionsschaltern.

TListBox
TListBox ermöglicht die Erstellung von und Auswahl aus Windows-Listen.

TComboBox
TComboBox ist von *TListBox* abgeleitet und definiert das Verhalten und die Eigenschaften von Windows-Kombinationsfestern. Ein Kombinationsfenster ist eine Kombination aus Liste und Editierfeld.

TCheckBox
TCheckBox regelt die Erstellung und die Verwaltung von Windows-Auswahlfeldern.

TRadioButton
TRadioButton regelt die Erstellung und Verwaltung von Windows-Schaltfeldern.

TGroupBox
TGroupBox regelt die Erstellung von Windows-Gruppen.

TEdit
TEdit unterstützt die leistungsstarken Textbearbeitungsmöglichkeiten für Windows-Editierfelder.

TScrollBar
Dieser Typ unterstützt die Erstellung und Bearbeitung einer selbständigen Bildlaufleiste.

TStatic
TStatic stellt Methoden zur Verfügung, die den Text eines statischen Steuerelements (nur Ausgabe) setzen, abfragen und löschen.

MDI-Objekte

Windows implementiert einen Standard für die Behandlung von mehreren Dokumenten im Rahmen eines einzelnen Fensters, der MDI (Multidokumentschnittstelle) genannt wird. ObjectWindows stellt Mittel für den Aufbau und die Manipulation von MDI-Fenstern zur Verfügung. MDI-Objekte werden in Kapitel 13 »MDI-Objekte« detailliert erläutert.

TMDIWindow
TMDIWindow sorgt dafür, daß das Verhalten und die Eigenschaften des Hauptfensters einer Anwendung entsprechen, die die MDI-Spezifikation von Windows befolgt.

TMDIClient
TMDIClient unterstützt zusätzlich MDI-Fenster. Das MDI-Client-Objekt ist das Objekt, das tatsächlich den MDI-Client-Bereich des Fensters bearbeitet, also den Bereich, der die untergeordneten Fenster (»Dokumente«) enthält.

ObjectWindows-Units

ObjectWindows implementiert die im vorhergehenden aufgeführten Typen in compilierten Turbo-Pascal-Units. Der vorliegende Abschnitt gibt einen Überblick über den Inhalt der Units. Nur die *WObject*-Unit muß in jeder ObjectWindows-Anwendung vorhanden sein.

WObject

WObject muß in jeder ObjectWindows-Anwendung eingesetzt werden. Diese Unit definiert die Standardtypen für Fensterobjekte, abstrakte Objekte, wie *TObject*, und weitere Typen, die Kollektionen und Streams implementieren.

WinProcs

WinProcs definiert Funktions- und Prozedurköpfe für die Windows-API (Applications Programming Interface). Über *WinProcs* kann auf jede in den Windows-Standardbibliotheken vorhandene Routine zugegriffen werden. Zusammen mit *WinTypes* definiert *WinProcs* die Turbo-Pascal-Implementierung der Windows-API. Die Unit *WinProcs* und ihr Inhalt werden im Abschnitt »Windows-Funktionen« weiter hinten in diesem Kapitel erläutert.

WinTypes

WinTypes definiert Turbo-Pascal-Versionen für alle Typen, die von Routinen der Windows-API verwendet werden, inklusive einfacher Typen und Datenstrukturen (Records) und aller Standard-Windows-Konstanten, inklusive Stile, Botschaften und Flags.

StdDlgs

StdDlgs definiert zwei Standarddialogtypen: *TFileDialog* und *TInputDialog*.

StdWnds

StdWnds definiert zwei Standardfenstertypen: *TDlgWindow* und *TFileWindow*.

Windows-Ressourcen

ObjectWindows beinhaltet zwei Ressourcendateien:

STDDLGS.RES, in der die für die *TFileDialog*- und *TInputDialog*-Objekte notwendigen Ressourcen stehen, und

STDWNDS.RES, die die von *TFileWindow*- und *TEditWindow*-Objekten verwendeten Ressourcen enthält.

Die Ressourcen werden mit einbezogen, wenn die entsprechenden Units verwendet werden, so daß jedes Programm, das die Unit *StdDlgs* einsetzt, Zugriff auf die Ressourcen in STDDLGS.RES hat.

Windows-Funktionen

Die Funktionsfähigkeit von Windows beruht auf den ca. 600 zur Verfügung stehenden Funktionen. Jede Funktion hat einen Namen, wie beispielsweise *LoadMenu* oder *MessageBox*. Die einzige Möglichkeit für eine Windows-Anwendung, die Umgebung zu manipulieren, die Anzeige (Darstellungsform) zu ändern oder als Antwort auf Anwendereingaben zu reagieren, besteht im Aufruf einer oder mehrerer Windows-Funktionen. Mit ObjectWindows können Sie jedoch Fenster anlegen, Dialogfenster anzeigen und Steuerelemente manipulieren, ohne eine Windows-Funktion dazu aufrufen zu müssen. Wie ist dies möglich?

ObjectWindows-Aufrufe von Windows-Funktionen

Die Methoden von ObjectWindows rufen Windows-Funktionen auf. Dennoch kopiert ObjectWindows nicht deren Funktionalität, es faßt vielmehr die Liste der Windows-Funktionen und die API in einer dynamischen, objektorientierten Bibliothek erneut zusammen. ObjectWindows vereinfacht spürbar die für Windows-Funktionen notwendige Angabe zahlreicher Parameter. Häufig stellt ObjectWindows selbstständig Parameter zur Verfügung, wie beispielsweise Fenster-Handles und die IDs untergeordneter Fenster, die als Felder in Schnittstellenobjekten gespeichert sind.

So erfordern viele Windows-Funktionen beispielsweise ein Handle auf ein

Fenster, mit dem das betreffende Fenster identifiziert wird. Diese Funktionen werden normalerweise von den Methoden eines Fensterobjekts aufgerufen. Das Objekt führt das Handle, das mit seinem Fenster verknüpft ist, im Feld *HWindow*. Dieses Feld kann als Handle übergeben werden, so daß Sie es nicht jedesmal selber angeben müssen. Auf diese Weise dienen die Objekttypen von ObjectWindows als objektorientierte Schicht, die mit Aufrufen für die nicht objektorientierte Windows API implementiert ist.

Zugriff auf Funktionen

Um von ObjectWindows-Anwendungen aus direkt auf die Windows-Funktionen zugreifen zu können, müssen Sie die Unit *WinProcs* verwenden. *WinProcs* definiert für jede Windows-Funktion einen Turbo-Pascal-Prozedur- oder -Funktionskopf, mit deren Hilfe Sie Windows-Funktionen genauso wie eine Turbo-Pascal-Routine aufrufen können. In Kapitel 2 »Windows-Funktionen« im *Windows-Referenzhandbuch* oder in der Datei WINPROCS.INT finden Sie eine Liste der Funktionsköpfe.

Bei Windows-Funktionen müssen verschiedene Konstanten vom Typ Word und Longint als Argumente übergeben werden. Diese Konstanten repräsentieren Fenster, Dialogfenster und Steuerelementstile ebenso wie Rückgabewerte und vieles mehr. Code, der diese Konstanten einsetzt, ist besser lesbar, einfacher zu handhaben und unabhängiger von Änderungen in späteren Windows-Versionen als Code, der Zahlen verwendet. So ist es beispielsweise informativer, ein Fenster mit dem Stil *ws_Popup* als mit $80000000 zu definieren. *ws_Popup* und die anderen Konstanten sind in der Unit *WinTypes* definiert und werden in Kapitel 1 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch* erläutert.

Einige Funktionen machen komplexere Datenstrukturen erforderlich, wie zum Beispiel Schriften beschreibende Strukturen (*TLogFont*) oder Fensterklassen (*TWndClass*). Windows und ObjectWindows definieren diese und weitere Datenstrukturen.

Eine Liste der verfügbaren Strukturen finden Sie in Kapitel 4 »Windows-Typen« im *Windows-Referenzhandbuch* oder in der Datei WINTYPES.INT.

Bei der Verwendung von ObjectWindows sind alle Windows-Funktionen direkt verfügbar und können von Ihrem Quellcode aus aufgerufen werden, wenn *WinProcs* in der uses-Anweisung eingesetzt wird. So ruft beispielsweise der nachfolgende Code die Windows-Funktion *MessageBox* zur Erstellung eines Meldungsfensters auf:

```
Reply := MessageBox (HWindow 'Do you want to save?', 'File has chan-
ged', mb_YesNo or mb_IconQuestion);
```

Der von *MessageBox* zurückgelieferte Wert ist eine Integerzahl, deren Wert die Anwenderaktion angibt, die für das Schließen des Meldungsfensters vorgenommen wurde. Wenn der Anwender den Schalter Yes angeklickt hat, entspricht das Ergebnis der Windows-definierten Integerkonstante *id_Yes*. Wenn der Anwender den Schalter No angeklickt hat, ist das Ergebnis *id_No*.

Kombinieren von Stilkonstanten

Windows-Funktionen, die Schnittstellenelemente produzieren, verlangen normalerweise Stilparameter vom Typ Word oder Longint. Windows definiert hunderte von Stilkonstanten, die Sie in Kapitel 1 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch* finden. Bezeichner für Stilkonstanten bestehen aus einem zweibuchstabigen Präfix, gefolgt von einem Unterstrich und einem (beschreibenden)Namen. So ist *ws_Popup* zum Beispiel eine Fensterstilkonstante für Pop-up-Fenster, wobei *ws_* »window style« bedeutet.

Diese Stile müssen zur Erzeugung eines anderen Stils häufig kombiniert werden. Im dem *MessageBox*-Beispiel übergeben Sie *mb_YesNo* oder *mb_IconQuestion* als Stilparameter. Dieser Stil produziert ein Meldungsfenster mit den beiden Schaltern Yes und No und ein Fragezeichensymbol. Der bitweise or-Operator kombiniert die beiden Konstanten Bit für Bit. Der daraus entstehende Stil ist eine Kombination beider Stile.

Denken Sie aber daran, daß sich einige Stile gegenseitig ausschließen. Durch die Kombination unverträglicher Stile kann es zu unvorhersehbaren und unerwünschten Ergebnissen kommen.

Typen von Windows-Funktionen

Im folgenden finden Sie die Arten von Windows-Funktionen, die für Ihre ObjectWindows-Programme zur Verfügung stehen.

Schnittstellenfunktionen der Fensterverwaltung
Diese Funktionen bearbeiten Botschaften, manipulieren Fenster und Dialogfenster und erstellen Systemausgaben. Die Kategorie umfaßt Funktionen für Menüs, Mauszeiger und die Zwischenablage.

Funktionen der grafischen Geräteschnittstelle (GDI)
Diese Funktionen geben Text, Grafiken und Bitmaps auf eine Vielzahl verschiedener Geräte, einschließlich Bildschirm und Drucker, aus. Die Funktionen sind nicht an ein bestimmtes Gerät gebunden, sie sind geräteunabhängig.

Schnittstellenfunktionen der Systemdienste
Diese Funktionen handhaben eine Vielzahl von Systemdiensten, einschließlich Speicherverwaltung, Schnittstelle zum Betriebssystem, Ressourcenverwaltung und Kommunikationsvorgänge.

Callback-Funktionen

Windows verfügt über Auflistfunktionen, mit deren Hilfe Sie in einer Schleife bestimmte Elementtypen im Windows-System auflisten können. So könnten

Sie beispielsweise die Schrift im System auflisten und für jede Schrift ein Textbeispiel drucken. Zur Verwendung dieser Funktionen müssen Sie einen Zeiger auf eine Funktion (die *Callback*-Funktion) übergeben, die Windows dann während des Auflistprozesses aufruft. Die angegebene Funktion wird direkt von Windows aufgerufen, wenn Fenstereigenschaften, Fenster, untergeordnete Fenster, Schriften und ähnliches aufgelistet werden.

Die folgenden Windows-Funktionen erfordern Callback-Funktionen: *CreateDialog, CreateDialogIndirect, DialogBox, DialogBoxIndirect, EnumChildWindows, EnumClipboardFormats, EnumFontFunc, EnumMetaFile, EnumObjects, EnumProps, EnumTaskWindows, EnumWindows, GlobalNotify, GreyString, LineDDA, RegisterClass, SetResourceHandler, SetTimer, SetWindowsHook*

Sie können einen Zeiger auf eine virtuelle Methode als Callback-Funktion übergeben. Beachten Sie aber, daß dann immer diese Methode aufgerufen wird, auch dann, wenn sie überschrieben wurde. Mit anderen Worten: Die Adresse wird zur Compilierzeit aufgelöst, so daß immer eine bestimmte Methode aufgerufen wird. In den meisten Fällen sollte eine statische Methode oder eine reguläre Funktion verwendet werden. Der Zeiger auf die Funktion wird als der erste Parameter vom Typ *TFarProc* in diesen Methoden übergeben. Wenn Sie beispielsweise die Pascal-Callback-Funktion *ActOnWindow* folgendermaßen definiert haben:

```
function ActOnWindow (TheHandle: HWnd; TheValue: Longint): Integer;
```

können Sie sie als Callback in einem Aufruf der Windows-Funktion *EnumWindows* übergeben:

```
ReturnVal := EnumWindows (TFarProc(ActOnWindows), ALongint);
```

Parameter und Rückgabewert einer Callback-Funktion müssen genau von dem Typ sein, den die aufrufende Windows-Funktion erwartet. Die Funktion *ActOnWindow* würde Aktionen in dem vom übergebenen Handle bestimmten Fenster ausführen. Der Parameter *TheValue* ist ein beliebiger Wert, den der Aufrufer von *EnumWindows* zur Übergabe auswählt.

Windows-Botschaften empfangen

Eine typische Windows-Anwendung führt mehr aus, als nur die Verteilung und Bearbeitung von Botschaften. Wenn der Anwender beispielsweise die Maus in einem Fenster anklickt, sendet Windows eine Botschaft an die Anwendung.

So wie es verschiedene Windows-Funktionen gibt, gibt es auch unterschiedliche Windows-Botschaften, die alle einen eigenen Namen haben, wie zum Beispiel *wm_LButtonDown* oder *wm_Command* (wobei *wm_* für »Windows-Message« steht). Eine Windows-Botschaft ist wie ein Funktionsaufruf, in dem Parameter befördert werden, die mit wichtigen Daten angefüllt sind. Auf diese Weise ist eine Botschaft die Art und Weise, in der Windows eine Funktion in Ihrem Programm aufruft.

Eine Anwendung ist für den Empfang und für das Verteilen von Botschaften

an den zuständigen Programmteil verantwortlich. So könnte eine Windows-Anwendung zum Beispiel eine Routine definieren, die den Fall bearbeitet, in dem ein Anwender in einem Fenster mit der Maus klickt. ObjectWindows verteilt die Botschaft mit Hilfe eines Zusatzes zur virtuellen Methodentabelle von Turbo Pascal (VMT), der sogenannten dynamischen Methodentabelle (DMT). Die DMT ermöglicht es Ihnen, eine Methodendefinition um einen Bezeichner zu erweitern, der einer Windows-Botschaft entspricht. Sie müssen dazu nur Methoden schreiben, die Windows-Botschaften beantworten. Wir nennen diese Methoden *Anwortmethoden*.

Im folgenden sehen Sie einen Definitionskopf für eine Antwortmethode, die aufgerufen wird, wenn der Anwender den linken Mausknopf betätigt:

```
procedure WMButtonDown(var Msg: TMessage); virtual wm_First +
  wm_LButtonDown;
```

Wie Ereignisse Methodenaufrufe werden

Wenn Sie erfolgreich mit ObjectWindows programmieren wollen, müssen Sie sich mit dem Schreiben von Antwortmethoden vertraut machen. Dazu müssen Sie wiederum wissen, welche Ereignistypen welche Botschaftstypen ergeben. Folgende Vorgänge produzieren Windows-Botschaften:

- Der Anwender erzeugt durch Bewegen oder Betätigen der Maus oder durch Eingaben über die Tastatur eine Anwenderereignis-Botschaft.
- Ihr Programm kann Botschaften direkt an sich selbst senden.
- Ihr Programm kann Windows-Funktionen aufrufen, die wiederum andere Windows-Botschaften empfangen.
- Anwendungen können Botschaften durch dynamischen Datenaustausch (DDE) an andere Anwendungen senden.
- Die Windows-Shell, einschließlich des Programm-Managers, sendet eine Botschaft an alle geöffneten Anwendungen, bevor sie geschlossen werden.

Die Beantwortung von Anwenderereignis-Botschaften ist das wichtigste Mittel für die Interaktion mit dem Anwender. Wir bezeichnen dieses Vorgehen als ereignisorientierte Programmierung. Wenn ein Ereignis eintritt, erzeugt das Windows-Modul USER.EXE eine Botschaft und bringt sie in die *Botschaftschlange* der Anwendung. Obwohl es für jede Windows-Sitzung nur eine USER.EXE gibt, hat jede aktive Anwendung ihre eigene Botschaftschlange. Jede Anwendung muß sich um ihre Botschaftschlange und die darin enthaltenen Botschaften kümmern. So erledigt Windows Multitasking-Aufgaben. Sehen Sie sich dazu die Abbildung 7.2 an.

In jeder Windows-Anwendung gibt es ein Codefragment, das *Botschaftschleife* genannt wird, und das während der Programmoperation abläuft. Die Botschaftschleife kümmert sich um die Botschaftschlange und verteilt die Botschaften im Programm. Eine ObjectWindows-Botschaftschleife ist die *MessageLoop*-Methode des *TApplication*-Objekts.

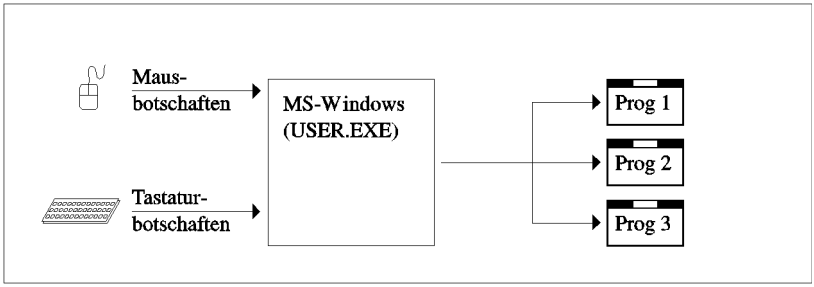


Abbildung 7.2 Die Übergabe einer Botschaft an eine Anwendung

Jede Botschaft von Windows bezeichnet das mit dem auslösenden Ereignis am engsten verbundene Fenster, indem das Handle des Fensters in den Botschafts-Record aufgenommen wird. ObjectWindows leitet die Botschaft an das Fensterobjekt weiter, das mit diesem Fenster-Handle verknüpft ist. ObjectWindows bestimmt auch, ob dieses Fenster eine Antwortmethode für die Botschaft definiert, und wenn dies der Fall ist, ruft ObjectWindows diese Methode auf. Andernfalls wird die Standardantwort aktiviert. So ist sichergestellt, daß alle Botschaften auf jeden Fall beantwortet werden.

Wenn der Anwender beispielsweise in einem Fenster mit der Maus klickt, enthält die sich daraus ergebende Botschaft ein Handle auf dieses Fenster. Ein Fensterobjekt empfängt also Botschaften, für die sein Typ Anwortmethoden definiert oder nicht. Ist keine Antwortmethode definiert, aktiviert ObjectWindows die voreingestellte Methode *DefWndProc* des Objekts. Näheres zur Bearbeitung von Standardbotschaften finden Sie in Kapitel 9 »Schnittstellenobjekte«.

Die Parameter einer Windows-Botschaft

Jede Windows-Botschaft hat ein Handle auf eine Botschaftsstruktur mit sechs Feldern. Zwei dieser Felder werden intern von Windows verwendet. Zwei Felder, das Fenster-Handle und der Botschafts-Bezeichner, benützt ObjectWindows zur Weiterleitung der richtigen Botschaft an das richtige Fenster. Nur zwei Parameter enthalten Daten, die direkt für Ihr Programm nützlich sind.

wParam ist vom Typ Word und beinhaltet Informationen wie Handle, Bezeichner oder Code. Beispielsweise erzeugt eine Menüauswahl eine *wm_Command*-Botschaft, deren Parameter *wParam* der ID des ausgewählten Menüpunktes entspricht.

lParam ist vom Typ Longint und beinhaltet normalerweise zwei Arten von Informationen, eine im niederwertigen Word und eine im höherwertigen Word. *lParam* führt komplexere Daten, wie Koordinatenpunkte oder Zeiger auf Strings (PChar). Die Botschaft *wm_LButtonDown*, die bei Betätigen der linken Maustaste im Fenster erzeugt wird, führt die X-Koordinate des angeklickten Punktes im niederwertigen Wort und die Y-Koordinaten im höherwertigen Wort von

lParam. Da *TMessage* als varianter Record definiert ist, kann auf die höher- und niederwertigen Worte als *lParamHi* und *lParameLo* zugegriffen werden.

Typen von Windows-Botschaften

Im folgenden finden Sie eine Zusammenstellung der Botschaftstypen, die eine Windows-Anwendung empfangen kann.

Botschaften der Fensterverwaltung
Diese Botschaften signalisieren, daß der Status eines Fensters geändert wurde. So wird zum Beispiel *wm_Close* gesendet, wenn ein Fenster geschlossen wird oder *wm_Paint*, wenn ein Teil eines Fensters oder ein gesamtes Fenster erneut angezeigt werden muß.

Initialisierungsbotschaften
Diese Botschaften werden gesendet, wenn ein Programm ein Menü (*wm_InitMenu*) oder ein Dialogfenster (*wm_InitDialog*) erzeugt.

Eingabebotschaften
Diese Botschaften werden als das Ergebnis einer Eingabe mit der Maus (*wm_MouseMove*), über die Tastatur (*wm_Char*), die Bildlaufleisten (*wm_VScroll*) oder die Systemuhr (*wm_Timer*) empfangen. Eine der häufigsten Eingabebotschaften ist *wm_Command*, die von einer Menü- oder Accelerator-Auswahl bzw. von einem Ereignis, wie dem Anklicken eines Schalters, stammt.

Systembotschaften
Diese Botschaften werden als Antwort auf Anwendermanipulationen am Standardzubehör der Anwendung gesendet, wie beispielsweise am Systemmenüsymbol, den Bildlaufleisten oder bei Betätigung der Schalter für Minimalisierung oder Maximalisierung. *wm_SysCommand* wird als Ergebnis einer Auswahl aus dem Systemmenü gesendet. Die meisten Anwendungen fangen keine Systembotschaften ab.

Zwischenablagebotschaften
Diese Botschaften werden gesendet, wenn eine andere Anwendung auf die Zwischenablage Ihres Fensters zuzugreifen versucht.

Systeminformationsbotschaften
Diese Botschaften werden gesendet, wenn ein Attribut auf der Windows-Systemebene geändert wird, wie beispielsweise die Farbe (*wm_SysColorChange*) oder die Schrift (*wm_FontChange*).

Botschaften der Manipulation von Steuerelementen
Die Anwendung sendet diese Botschaften an ihre Steuerelemente, beispielsweise an Listen und Schalter. Jeder Steuerelementtyp versteht seine spezifischen Botschaften für das Abfragen und Bestimmen von Attributen. Steuerelement-

botschaften unterscheiden sich von allen anderen Botschaften – mit Ausnahme von MDI-Botschaften – dadurch, daß sie die Anwendung nicht von einem Ereignis *benachrichtigen*, sondern ein Ereignis *bewirken*. Lesen Sie dazu »Senden von Botschaften« weiter hinten in diesem Kapitel.

Benachrichtigungsbotschaften von Steuerelementen
Diese Botschaften sind *wm_Command*-Botschaften, die ein übergeordnetes Fenster von einem Steuerelementereignis in Kenntnis setzen, zum Beispiel wenn ein Eintrag aus einer Liste ausgewählt oder etwas in ein Editierfeld eingegeben wird. Die Parameter dieser Botschaften bestimmen einen besonderen Benachrichtigungscode, wie beispielsweise *lbn_SelChange* oder *en_Change*.

Benachrichtigungsbotschaften von Bildlaufleisten
Diese Botschaften sind spezielle Benachrichtigungsbotschaften für Bildlaufleisten. Es gibt zwei Botschaften dieser Art: *wm_HScroll* und *wm_VScroll*.

Botschaften von Nicht-Client-Bereichen
Diese Botschaften ähneln den Eingabebotschaften, aber beziehen sich speziell auf Ereignisse außerhalb des Client-Bereichs des Fensters, einschließlich Menü-, Titelzeile und Rahmen.

Es gibt zwei Botschaften dieser Art: *wm_NCMouseMove* und *wm_NCPaint*. Normalerweise benötigt Ihre Anwendung diese Botschaften nicht.

MDI-Botschaften (Botschaften der Multidokumentschnittstelle)
Diese Botschaften werden von einer Anwendung gesendet, die dem MDI-Standard zur Manipulation ihrer untergeordneten Fenster folgt. Die MDI-Botschaften sind: *wm_MDIActivate* und *wm_MDIDestroy*.

Standardbearbeitung von Botschaften

Einige Windows-Botschaften *erfordern* eine bestimmte Standardaktion oder einen bestimmten Rückgabewert. Die Windows-Funktion *DefWindowProc* stellt die Standardaktion für jede Botschaft zur Verfügung. ObjectWindows ruft *DefWindowProc* bei ignorierten Botschaften auf. Das Abfangen dieser Botschaften mit Antwortmethoden kann jedoch notwendige Aktionen außer Kraft setzen. Dies gilt aber nur für einige Windows-Botschaften. In diesen Fällen sollten Sie *DefWindowProc* direkt (über die *TWindowsObject*-Methode *DefWndProc*) aufrufen.

Senden von Botschaften

Eine Quelle für Botschaften ist Ihr Programm selbst. Ihr Programm kann die Windows-Funktion *SendMessage* aufrufen. Als Ergebnis empfängt Ihr Programm

eine Botschaft. Dieses Vorgehen ist dann hilfreich, wenn ein Ereignis von Ihrem Programm simuliert werden soll.

SendMessage sollte vor allen zur Manipulation von Steuerelementen, wie Listen und Schaltern, eingesetzt werden. Windows definiert *Botschaften von Steuerelementmanipulationen*, um zum Beispiel Einträge in eine Liste einzufügen (*lb_AddString*) oder den Status eines Auswahlfeldes zu ändern (*bm_SetCheck*). Die Steuerelementobjekte von ObjectWindows definieren Methoden, wie zum Beispiel *TListBox.AddString* und *TCheckBox.SetCheck*, die viele dieser häufig verwendeten Botschaften an die mit ihnen verknüpften Schnittstellenelemente senden. Wenn Sie eine seltener verwendete Botschaft an die Steuerelemente senden möchten, verwenden Sie *SendMessage*, wobei *HWindow* des Steuerelements der erste Parameter (*Wnd*) sein soll.

Die Vorgehensweise für das Senden einer Botschaft an ein Steuerelement in einem Dialogfenster gestaltet sich etwas anders. Häufig sind den Steuerelementen in Ihren *TDialog*-Objekten keine Schnittstellenobjekte zugeordnet. Sie können die Botschaften aber mit Hilfe der *SendDlgItemMsg*-Methode von *TDialog* an sie senden. Die Kommunikation mit den Steuerelementen in Dialogfenstern läßt sich vereinfachen, wenn diese Elemente mit Hilfe des *InitResource*-Konstruktors mit Schnittstellenelementen verknüpft werden. Näheres dazu finden Sie in Kapitel 11 »Dialogobjekte« im Abschnitt »Verknüpfen von Steuerelementobjekten«.

Botschafts-bereiche

Da Botschaften durch Werte vom Typ Word definiert werden, kann es 65535 Botschaften geben. Windows teilt Botschaften in eindeutige Bereiche für Standard-Windows-Botschaften, Befehlsbotschaften, Benachrichtigungsbotschaften, etc. ein. Um diese Bereiche leichter identifizieren zu können, definiert ObjectWindows Konstanten, die als Offsets für die einzelnen Bereiche verwendet werden.

Die Botschaftsbereiche und die ihnen entsprechenden ObjectWindows-Konstanten finden Sie in der Tabelle 7.1

Tabelle 7.1 Bereiche der Windows-Botschaften

Konstante	Wert	Bereich	Bedeutung
wm_First	$0000	$0000-$03FF	Fensterbotschaften
wm_User	$0400	$0400-$7FFF	Anwenderdefinierbare Botschaften
id_First	$8000	$8000-$8EFF	Botschaften der untergeordneten Fenster-ID
id_Internal	$8F00	$8F00-$8FFF	Reserviert für internen Gebrauch
nf_First	$9000	$9000-$9EFF	Benachrichtigungsbotschaften
nf_Internal	$9F00	$9F00-$9FFF	Reserviert für internen Gebrauch
cm_First	$A000	$A000-$FEFF	Befehlsbotschaften
cm_Internal	$FF00	$FF00-$FFFF	Reserviert für internen Gebrauch

Anwenderdefinierte Botschaften

Windows ermöglicht es Ihnen, eigene Botschaften für den Einsatz in Ihrer Anwendung zu definieren. Anwenderdefinierte Botschaften sind vor allem bei der Definition und Beantwortung neuer Ereignisse sehr hilfreich. So könnten Sie beispielsweise ein Fenster eine anwenderdefinierte Botschaft an alle anderen offenen Fenster in der Anwendung senden lassen.

Mit jeder Windows-Botschaft ist ein Bezeichner vom Typ Word verknüpft. Für vordefinierte Windows-Botschaften definiert ObjectWindows eine entsprechende Konstante vom Typ Word. Diese Konstante wird dann im Methodenkopf einer Botschaftsantwort-Methode eingesetzt.

Im zugelassenen Bereich für Botschaftsbezeichner sind Werte für anwenderdefinierte Botschaften reserviert. Der Wertebereich, der für vordefinierte Windows-Botschaften reserviert ist, liegt zwischen 0 und *wm_User* -1 (*wm_User* ist eine Konstante). Der Wertebereich für anwenderdefinierte Botschaften liegt zwischen *wm_User* und *wm_User* + 31744. Es ist am günstigsten, wenn anwenderdefinierte Werte als Konstanten definiert werden, die mit den Werten *wm_User*, *wm_User* + 1, *wm_User* + 2, usw. beginnen.

Sehen Sie sich das folgende Beispiel an:

```
const
wm_MyFirstMessage  = wm_User;
wm_MySecondMessage = wm_User + 1;
wm_MyThirdMessage  = wm_User + U2;
```

Senden Sie Ihre Botschaften dann mit der Windows-Funktion *SendMessage*:

```
SendMessage (AWindow^.HWindow, wm_MyFirstMessage, AWord, ALongint);
```

Für den Empfang einer anwenderdefinierten Botschaft gehen Sie wie bei Standard-Windows-Botschaften vor, zum Beispiel:

```
MyWindow = object (TWindow)

  ...

procedure WMMyFirstMessage (var Msg: TMessage); virtual
wm_First +  wm_MyFirstMessage;
procedure WMMySecondMessage (var Msg: TMessage); virtual
wm_First + wm_MySecondMessage;
procedure WMMyThirdMessage (var Msg: TMessage); virtual
wm_Fist +  wm_MyThirdMessage;
end;
```

8. Anwendungsobjekte

Die wichtigste Anforderung an eine ObjectWindows-Anwendung ist die Definition eines Anwendungsobjekts, das vom abstrakten Typ *TApplication* abgeleitet ist.

Das Anwendungsobjekt umfaßt folgende Verhaltensweisen und Eigenschaften einer ObjectWindows-Anwendung:

- Erstellen und Anzeigen des Hauptfensters der Anwendung.
- Initialisieren aller Instanzen einer Anwendung, beispielsweise zum Laden einer Accelerator-Tabelle. (Sie *können* viele Instanzen derselben ObjectWindows-Anwendung gleichzeitig laufen lassen.)
- Initialisieren der ersten Instanz einer Anwendung für Aufgaben, die allen Instanzen der Anwendung dienen, wie zum Beispiel das Konfigurieren einer Schnittstelle.
- Bearbeiten der Windows-Botschaften, die eine Anwendung empfängt.
- Schließen der Anwendung.

Neben der Definition eines Typs, der von *TApplication* abstammt, *müssen* Sie die Fähigkeit zur Erzeugung des Hauptfensterobjekts hinzufügen. Sie haben auch die Möglichkeit, die voreingestellten Verhaltensweisen bei der Initialisierung von Instanzen, dem Schließen der Anwendung und dem Bearbeiten von Windows-Botschaften neu zu definieren.

Steuerung des Anwendungsflusses

Im folgenden sehen Sie ein minimales Hauptprogramm in einer ObjektWindows-Anwendung:

```
program MinimalApp;
uses WObjects;
var
  MyApp: TApplication;
begin
  MyApp.Init('TestApp');
```

```
  MyApp.Run;
  MyApp.Done;
end.
```

Das Programm *MinimalApp* stellt den Minimalumfang einer ObjectWindows-Anwendung dar, in der keine neuen Objekttypen definiert werden müssen. Normalerweise werden Sie jedoch neue Objekttypen zumindest für die Anwendung und das Hauptfenster festlegen.

Das Anwendungsobjekt wird im Hauptprogramm einer ObjectWindows-Anwendung konstruiert. Falls erforderlich kann von anderen Teilen des ObjectWindows-Programms durch Bezugnahme auf die globale Variable *Application* darauf zugegriffen werden.

Das Hauptprogramm Ihrer ObjectWindows-Anwendung besteht normalerweise nur aus drei Anweisungen.

- Die erste Anweisung konstruiert das Anwendungsobjekt durch Aufruf des Konstruktors *Init*. *Init* initialisiert auch die Dateifelder des Anwendungsobjekts und ruft dann *InitApplication* und *InitInstance* auf, die die Initialisierung der ersten bzw. jeder anderen Instanz ausführen. Danach wird zur Erstellung des Hauptfensters *InitMainWindow* aufgerufen. Sobald *Init* beendet ist, wird das Hauptfenster Ihrer Anwendung auf dem Bildschirm angezeigt.

  In meisten Fällen müssen Sie nur *InitMainWindow* neu definieren (vgl. Abbildung 8.1).

- Die zweite Anweisung ruft die Anwendungsmethode *Run* auf. *Run* startet die Anwendung durch den Aufruf von *MessageLoop*. *MessageLoop* ruft Methoden auf, die mit der Bearbeitung der eingehenden Windows-Botschaften beginnen, also den Befehlen, die die Anwendungsaktivität steuern. *MessageLoop* ist genau das, was der Name zum Ausdruck bringt: Eine Botschaftschleife, die so lange abläuft, bis die Anwendung geschlossen wird.

- *Done* ist der Destruktor für Anwendungsobjekte. Durch ihn wird der vom Anwendungsobjekt belegte Speicherplatz vor Beendigung der Anwendung wieder freigegeben.

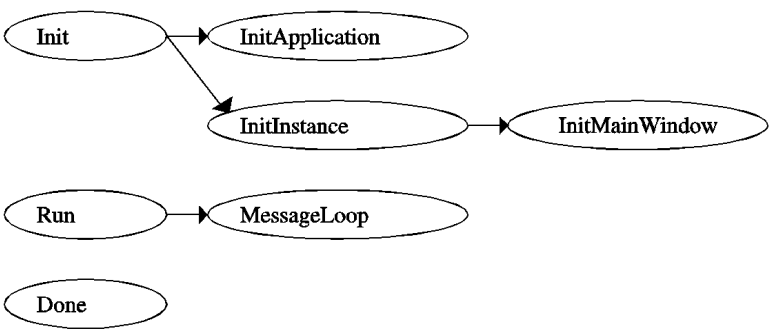


Abbildung 8.1 Methodenaufrufe, die den Anwendungsfluß steuern

Initialisieren von Anwendungen

Der Ablauf von Methodenaufrufen, die das Anwendungsobjekt initialisieren, ermöglicht es Ihnen, viele Teile des Prozesses selbst zu gestalten, indem Sie die Methoden neu definieren. Die einzige Methode, die Sie schreiben müssen, ist *InitMainWindow*. Sie können auch *InitInstance* zur Initialisierung jeder ausführenden Instanz einer Anwendung neu definieren, ebenso wie *InitApplication* zur Initialisierung der ersten ausführenden Instanz einer ObjectWindows-Anwendung.

Initialisieren des Hauptfensters

Im folgenden sehen Sie eine Minimaldefinition für den Objekttyp der Anwendung:

```
type
  MyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;
```

Sie müssen eine *InitMainWindow*-Methode definieren, die das Hauptfensterobjekt konstruiert und initialisiert und im Feld *MainWindow* des Anwendungsobjekts speichert. Im folgenden sehen Sie eine einfache *InitMainWindow*-Methode:

```
procedure MyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'The Main Window'));
end;
```

Diese Methode erzeugt eine neue Objektinstanz des ObjectWindows-Typs *TWindow* (*PWindow* ist ein Zeiger auf den Typ *TWindow*). Normalerweise wird Ihr Programm einen neuen Fenstertyp für sein Hauptfenster definieren und *InitMainWindow* wird diesen neuen Typ anstelle von *TWindow* verwenden.

Die obige einfache ObjectWindows-Anwendung besteht aus einem Hauptprogramm und der Definition des Anwendungstyps, der eine Methode, *InitMainWindow*, definiert. Im folgenden sehen Sie den Quelltext für das Programm, das sich vom früheren *MinimalApp* nur dadurch unterscheidet, daß sein Hauptfenster einen Titel hat:

```
program TestApp;

uses WObjects;

type
  MyApplication = object(TApplication)
    procedure InitMainWindow; virtual;
end;
```

```
procedure MyApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'The Main Window'));
end;

var
  MyApp: MyApplication;
begin
  MyApp.Init('TestApp');
  MyApp.Run;
  MyApp.Done;
end.
```

TestApp ist eine Minimalanwendung, die lediglich ein Fenster mit dem Titel 'The Main Window' anzeigt (vgl. Abbildung 8.2). Sie können dieses Fenster verschieben und in seiner Größe verändern. Sie können es zu einem Symbol verkleinern, indem Sie das Feld ↓ in der oberen rechten Ecke anklicken. Wenn Sie den ursprünglichen Zustand wieder herstellen wollen, klicken Sie das Symbol doppelt an. Wenn Sie das Feld ↑ anklicken, wird das Fenster auf die volle Bildschirmgröße vergrößert. Schließen Sie das Fenster und beenden Sie die Anwendung, indem Sie das Systemmenüsymbol in der oberen linken Ecke anklicken. *TestApp* ist eine voll funktionsfähige, rudimentäre Anwendung, die nur ein sehr einfaches Hauptfenster anlegt.

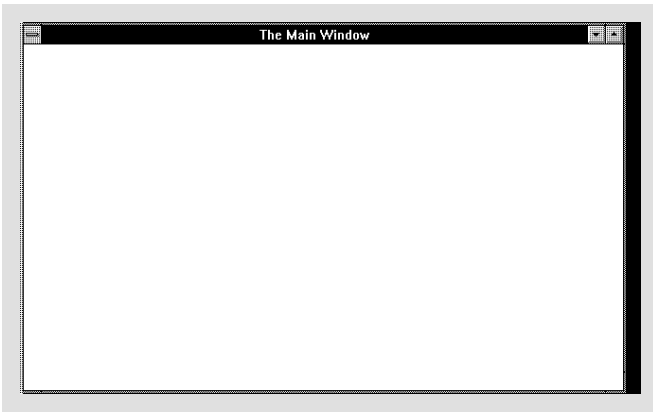


Abbildung 8.2
Das Hauptfenster

Initialisieren jeder Anwendungsinstanz

Der Anwender kann mehrere Instanzen einer ObjectWindows-Anwendung gleichzeitig ausführen. Die *InitInstance*-Methode initialisiert jede Instanz der Anwendung. Sie sollte nur eine Initialisierung der Anwendungsinstanz ausführen. Die Hauptfenster-Initialisierung sollte in *InitMainWindow* durchgeführt werden. *TApplication* definiert eine *InitInstance*-Methode, die *InitMainWindow* aufruft und dann das Hauptfenster mit einem Aufruf seiner *Show*-Methode

anzeigt. Sie müssen nur *InitInstance* neu definieren, um die Standardinitialisierung zu ändern, beispielsweise um eine Accelerator-Tabelle zu laden (eine anwendungsorientierte Aktivität). Wenn Sie *InitInstance* für Ihren Anwendungstyp neu definieren, müssen Sie sicherstellen, daß *TApplication.InitInstance* zuerst aufgerufen wird.

Im folgenden sehen Sie eine *InitInstance*-Methode, die eine Accelerator-Tabelle lädt, bevor die Anwendung gestartet wird. Die Konstante 100 bezeichnet die Accelerator-Tabelle, die in der Ressourcendatei definiert ist:

```
procedure TEditApplication.InitInstance;
begin
  TApplication.InitInstance;
  HAccTable := LoadAccelerators(HInstance, PChar(100));
end;
```

Initialisieren der ersten Anwendungsinstanz

Wenn der Anwender Ihre Anwendung mehrmals gleichzeitig ausführt (ohne sie zu beenden), können Sie einen Arbeitsschritt definieren, der nur beim ersten Start durchgeführt wird. Wir nennen dies die Initialisierung der ersten Instanz. Beachten Sie, daß jede Instanz als erste Instanz betrachtet wird, wenn der Anwender die Anwendung startet und beendet, wieder startet, etc.

Wenn die aktuelle Instanz eine erste Instanz ist, wird die Methode *InitApplication* aufgerufen. *TApplication* definiert eine Platzhalter-Methode *InitApplication*, die zur Ausführung der speziellen Initialisierung der ersten Instanz neu definiert werden kann.

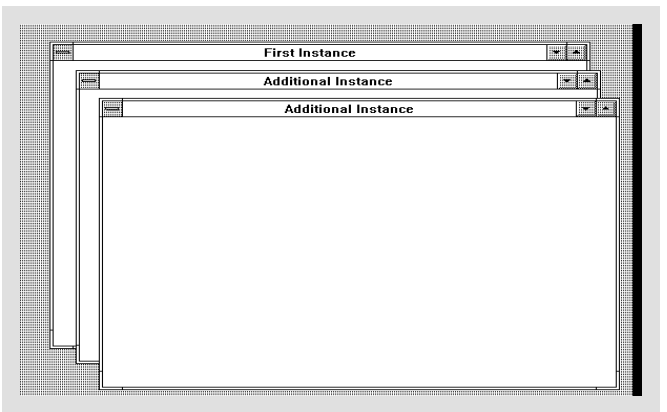


Abbildung 8.3 Neudefinieren der Anwendungsinitialisierung

Sie können zum Beispiel *TestApp* so ändern, daß der Titel des Hauptfensters anzeigt, ob es sich um die erste Instanz handelt oder nicht. Zu diesem Zweck fügen Sie ein Boolesches Objektfeld mit dem Namen *FirstApp* in den Anwendungstyp *MyApplication* ein. Dann definieren Sie eine *InitApplication*-Methode,

die *FirstApp* auf *True* setzt. Nur die erste Instanz der Anwendung wird den Wert *True* für *FirstApp* haben. Schließlich veranlassen Sie *InitMainWindow*, *FirstApp* zu überprüfen und einen entsprechenden Titel im Hauptfenster anzuzeigen. Sehen Sie sich dazu die Abbildung 8.3 an.

```
program TestApp;

uses WObjects;

type

MyApplication = object(TApplication)
  FirstApp: Boolean;
  procedure InitMainWindow; virtual;
  procedure InitApplication; virtual;
end;

procedure MyApplication.InitMainWindow;
begin
  if FirstApp then
  MainWindow := New(PWindow, Init(nil, 'First Instance'))
  else MainWindow := New(PWindow, Init (nil, 'Additional
                                        Instance));
end;

procedure MyApplication.InitApplication;
begin
  FirstApp := True;
end;

var
  MyApp: MyApplication;
begin
  MyApp.Init('TestApp');
  MyApp.Run;
  MyApp.Done;
end.
```

Anwendungen ausführen

Die Botschaftschleife der Anwendung wird aufgerufen, wenn das Programm seine Anwendungsobjektmethode *Run* aufruft, die wiederum ihre *MessageLoop*-Methode aktiviert. Während der Ausführung des Programms bearbeitet die Botschaftschleife die eingehenden Botschaften. Ihr ObjectWindows-Programm erbt eine *MessageLoop*-Methode, die automatisch arbeitet. Definieren Sie diese nur für spezielle Dialoge, Accelerator- oder MDI-Bearbeitung neu.

Die *MessageLoop*-Methode ruft zur Bearbeitung von Botschaften drei Übersetzungsmethoden auf. *ProcessDlgMessage* bearbeitet nicht-modale Dialoge, *ProcessAccels* bearbeitet Accelerator-Tasten und *ProcessMDIAccels* bearbeitet Ac-

celerator-Tasten für MDI-Anwendungen. Für Anwendungen, die keine Accelerator- Tasten oder nicht-modale Dialogfenster verwenden, bzw. die keine MDI-Anwendungen sind, können Sie *MessageLoop* vereinfachen. Näheres dazu finden Sie bei der Erläuterung der Übersetzungsmethoden bei *TApplication* in Kapitel 5 »Verzeichnis der Objekte« im *Windows-Referenzhandbuch.*

Anwendungen schließen

Der Aufruf von *Done* im Hauptprogramm der Anwendung gibt das Anwendungsobjekt frei. Voher muß jedoch die Botschaftschleife abgebrochen werden. Dies kann erfolgen, wenn der Anwender versucht, das Hauptfenster der Anwendung zu schließen. Wir haben gesagt, dies *kann* der Fall sein. Denn ObjectWindows verfügt über einen Mechanismus, der das Schließverhalten einer Anwendung neu definiert, beispielsweise vor dem Schließen auf nicht gespeicherte Dateien überprüft.

Im folgenden haben wir die Vorgänge zusammengestellt, die ablaufen, wenn der Anwender durch doppeltes Anklicken des Systemmenüsymbols die Anwendung schließen möchte:

1. Windows sendet eine *wm_Close*-Botschaft an das Hauptfenster der Anwendung.

2. Das Hauptfensterobjekt antwortet darauf mit einem Aufruf der *CanClose*-Methode des Anwendungsobjekts.

3. Die *CanClose*-Methode des Anwendungsobjekts ruft die *CanClose*-Methode des Hauptfensters auf.

4. Das Hauptfenster entscheidet, ob es sich selbst schließen wird. Voraussetzung dafür ist, daß alle untergeordneten Fenster das Schließen erlauben. Wenn alle *CanClose*-Methoden der untergeordneten Fenster *True* zurückliefern, liefert auch *CanClose* des Hauptfensters *True*, so daß die Anwendung geschlossen werden kann.

Alle Fensterobjekte erben eine *CanClose*-Methode, die den Wert *True* zurückliefert, um anzuzeigen, daß das Schließen der Anwendung erlaubt ist. Aus diesem Grund wurde die Anwendung *TestApp* ohne Verzögerung geschlossen. Sie können aber die *CanClose*-Methoden des Anwendungs- oder Hauptfenstertyps neu definieren und ein anderes Verhalten beim Schließen festlegen. Normalerweise werden Sie das Schließverhalten des Hauptfensterobjekttyps ändern, zum Beispiel so, daß vor dem Schließen auf nicht gespeicherte Dateien hin überprüft wird.

Dieser Mechanismus ist vergleichbar mit der Formel »Wenn es jemanden gibt, der weiß, warum die Anwendung nicht geschlossen werden sollte, möge er jetzt sprechen oder für immer schweigen«. Am Ende ist das Anwendungsobjekt für die Erlaubnis zum Schließen der Anwendung verantwortlich. Per Voreinstellung wird das Hauptfenster vor dem Schließen der Anwendung überprüft. Der Ablauf der Methodenaufrufe ist in Abbildung 8.4 skizziert.

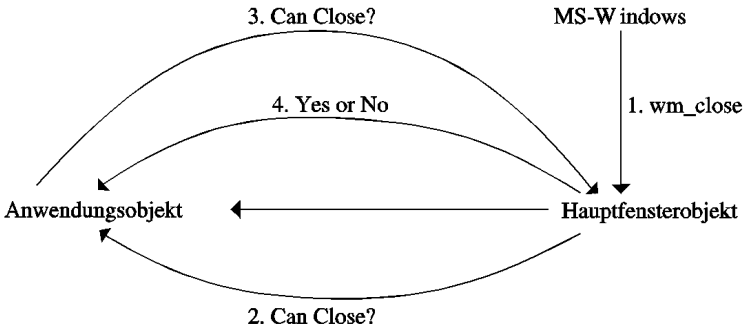


Abbildung 8.4 Methodenaufrufe zum Schließen einer Anwendung

Die *CanClose*-Methoden sind Boolesche Funktionen, von denen der Rückgabewert *True* erwartet wird, wenn die Anwendung geschlossen werden darf. Normalerweise führt eine modifizierte *CanClose*-Methode einige »Säuberungen« durch, bevor sie – bei erfolgreicher Bearbeitung – *True* zurückliefert.

Der Anwendungstyp eines ObjectWindows-Programms erbt die *CanClose*-Methode von *TApplication*, die lediglich die *CanClose*-Methode ihres Hauptfensterobjekts aufruft. Der Hauptfenstertyp erbt – außer bei Änderung seiner *CanClose*-Methode – *CanClose* von *TWindowObject*, die *True* zurückliefert, nachdem die *CanClose*-Methoden seiner untergeordneten Fenster aufgerufen wurden. Wenn Sie das Schließverhalten Ihres Hauptfensters ändern möchten, definieren Sie eine neue *CanClose*-Methode für Ihren Hauptfenstertyp. Diese Methode kann beispielsweise überprüfen, ob es offene Dateien gibt oder feststellen, ob der Anwender das Fenster wirklich schließen will.

9. Schnittstellenobjekte

Objekte zur Darstellung von Fenstern, Dialogfenstern und Steuerelementen werden als Schnittstellenobjekte bezeichnet. Dieses Kapitel befaßt sich

- mit den Anforderungen, die an ein Schnittstellenobjekt gestellt werden,
- mit dessen Verhaltensweise,
- mit dessen Beziehung zu den Fenstern, Dialogfenstern und Steuerelementen, die auf dem Bildschirm angezeigt werden und
- mit dem Verhältnis zwischen einzelnen Schnittstellenobjekten einer Anwendung und dem Mechanismus für die Übergabe von Windows-Botschaften.

Der Typ TWindowsObject

TWindowsObject ist ein abstrakter Objekttyp, in dem die abgeleiteten Windows-Schnittstellenobjekte vereinigt sind: *TDialog*, *TWindow* und dessen Nachkomme *TControl*. *TWindowsObject* definiert Verhaltensweisen, die allen Fensterobjekten, Dialogobjekten und Steuerelementobjekten gemeinsam sind.

Die Methoden von *TWindowsObject* erfüllen folgende Aufgaben:

- Verwaltung der dualen Objektstruktur von ObjectWindows und Windows. Dazu gehört auch die Erstellung und das Freigeben und Entfernen von Objekten.
- Unterstützung der hierarchischen Beziehungen zwischen den Schnittstellenobjekten (womit nicht deren objektbezogenen Vererbungsbeziehungen gemeint sind).
- Registrierung neuer Windows-Klassen (vgl. Kapitel 10 »Fensterobjekte«).

TWindowsObject arbeitet hinter den Kulissen, und Sie werden nur äußerst selten neue Typen direkt von *TWindowsObject* ableiten. *TWindowsObject* bildet die Basis für das objektorientierte ObjectWindows/Windows-Modell. Dieser Typ definiert einen großen Teil der Funktionsweise, die ein Objekt bei der Ableitung neuer Typen von *TWindow* und *TDialog* erbt. Das vorliegende Kapitel enthält eine Beispielanwendung, die eine neue Windows-Klasse registriert (dies kann beim Entwurf neuer Fenster- und Dialogtypen erforderlich sein).

Wofür Schnittstellenobjekte benötigt werden

Wofür werden nun Schnittstellenobjekte benötigt, wenn doch Windows bereits über visuelle Fenster, Dialogfenster und Steuerelemente verfügt?

Zu jedem Schnittstellenobjekt gehört normalerweise ein *visuelles Schnittstellenelement*. Das Schnittstellenobjekt ist das objektorientierte »Äquivalent« des jeweiligen sichtbaren Fensters, Dialogfensters oder Steuerelements. Das Schnittstellenobjekt stellt die Methoden zur Erstellung, Initialisierung, Verwaltung und zur Freigabe und Entfernung des zugehörigen Schnittstellenelements sowie Objektfelder zur Aufnahme von Daten – beispielsweise das entsprechende Handle und die über- und untergeordneten Fenster – zur Verfügung. Die Methoden der Schnittstellenobjekte nehmen Ihnen somit einen großen Teil der Arbeit bei der Programmierung für Windows ab.

Man könnte das Verhältnis zwischen Objekt und Element mit demjenigen einer DOS-Datei und einer Pascal-Dateivariablen vergleichen: Ebenso wie eine Dateivariable die physikalische Struktur der Datei repräsentiert, wird eine Variable C (ein Objekt) zur Repräsentation eines physikalischen Fensters, Dialogfensters oder Steuerelements definiert, das von der Windows-Fensterverwaltung verwaltet wird.

Bei der Strukturierung eines Schnittstellenobjekts und dem zugehörigen Schnittstellenelement muß auf die korrekte Behandlung von Fenstern, Dialogfenstern und Steuerelementen geachtet werden. So ist zum Beispiel für die Erstellung eines vollständigen Schnittstellenobjekts der Aufruf von *zwei* Methoden erforderlich. Die erste besteht aus einem Konstruktor (wie *Init*), der die Instanz des Schnittstellenobjekts konstruiert und – falls erforderlich – ihre Attribute setzt (beispielsweise den Stil und das Menü). Bei der zweiten Methode handelt es sich um die Erstellungsmethode *MakeWindow*, die das Schnittstellenobjekt mit einem neuen Schnittstellenelement verknüpft. Diese Verknüpfung wird durch das *HWindow*-Feld des Schnittstellenobjekts verwaltet, in dem ein Handle auf ein Fenster gespeichert ist (Handles auf Fenster werden in Kapitel 7 »Überblick über ObjectWindows« ausführlich beschrieben). *MakeWindow* ist eine Methode des Anwendungsobjekts und ruft die *Create*-Methode des Objekts auf. *Create* weist Windows an, das visuelle Element zu erstellen und ruft die *SetupWindow*-Methode auf, die das Schnittstellenobjekt initialisiert (beispielsweise für die Erstellung von untergeordneten Fenstern).

Normalerweise erhält ein neues Schnittstellenelement von Windows eine *wm_Create*-Botschaft, auf die es mit der Initialisierung antwortet. Schnittstellenobjekte von ObjectWindows erhalten keine *wm_Create*-Botschaften, deshalb müssen Sie selbst eine *SetupWindow*-Methode definieren.

Ähnlich ist es bei der Eliminierung des Schnittstellenobjekts: Sie müssen das visuelle Schnittstellenelement entfernen und das entsprechende Schnittstellenobjekt freigeben. Die Methode *Destroy* wird entweder direkt vom Programm oder indirekt durch das Schließen des Fensters (bzw. des übergeordneten Fensters) durch den Anwender aufgerufen und eliminiert das Schnittstellenelement. Schnittstellenelemente können auch entfernt werden, ohne das entsprechende Objekt freizugeben. Diese Vorgehensweise ist dann empfehlenswert,

wenn das Element später erneut erzeugt und angezeigt werden soll. Durch die Eliminierung des Schnittstellenelements wird das Handle des Objekts auf 0 gesetzt. Sie können das Handle überprüfen und so feststellen, ob ein Schnittstellenobjekt noch mit einem Schnittstellenelement verknüpft ist.

Die Erstellung eines Schnittstellenobjekts und des zugehörigen visuellen Schnittstellenelements führt nicht notwendigerweise zu einer Anzeige auf dem Bildschirm. Windows überprüft bei der Erstellung des visuellen Elements, ob für das Fenster der Stil *ws_Visible* gesetzt ist. Das Setzen und Löschen von *ws_Visible* und anderen Windows-Stilen erfolgt im *Attr.Style*-Feld des Objekts durch den *Init*-Konstruktor. Ist *ws_Visible* gesetzt, wird das Schnittstellenelement angezeigt, ist *ws_Visible* nicht gesetzt, wird es verborgen.

Ein Element kann jederzeit angezeigt oder verborgen werden, indem man die *Show*-Methode des Schnittstellenobjekts aufruft.

Das Verhältnis zwischen über- und untergeordneten Fenstern

In einer Windows-Anwendung sind Schnittstellenelemente (Fenster, Dialogfenster und Steuerelemente) über eine hierarchische Beziehung miteinander verbunden. Zwei Schnittstellenelemente sind miteinander verknüpft, wenn das eine das übergeordnete und das andere dessen untergeordnetes Fenster ist. Diese Beziehung darf nicht mit Vererbung oder mit dem Zugehörigkeitsverhältnis von Instanzen verwechselt werden, bei denen es sich um Objektbeziehungen handelt. Ein untergeordnetes Fenster erbt nichts von seinem übergeordneten Fenster und muß auch nicht in einem Objektfeld des übergeordneten Fensters gespeichert werden (obwohl dies oft der Fall sein wird).

Ein untergeordnetes Fenster ist ein Schnittstellenelement, das von einem anderen Schnittstellenelement verwaltet wird. Beispielsweise werden Listen von dem Fenster bzw. von dem Dialogfenster verwaltet, in dem sie enthalten sind. Sie werden nur dann angezeigt, wenn auch ihr übergeordnetes Fenster angezeigt wird. Dialogfenster sind untergeordnete Fenster und werden von dem Fenster verwaltet, das sie hervorbringt. Wird das übergeordnete Fenster geschlossen oder verschoben, so werden die untergeordneten Fenster automatisch geschlossen bzw. in manchen Fällen mit verschoben. Das Hauptfenster ist das oberste übergeordnete Fenster für alle untergeordneten Schnittstellenelemente einer Anwendung.

Hinweis: Nur Fenster und Dialogfenster können übergeordnete Fenster sein, nicht aber Steuerelemente. Dagegen kann jedes Schnittstellenelement (Dialogfenster, Fenster oder Steuerelement) ein untergeordnetes Fenster sein.

Listen untergeordneter Fenster

Das übergeordnete Fenster eines Schnittstellenelements muß bei dessen Konstruktion angegeben werden. Das Objekt des übergeordneten Fensters wird

als Parameter für den Konstruktor des Schnittstellenelements *Init* benötigt (ein Beispiel dafür finden Sie in Kapitel 10 »Fensterobjekte«). Für den Zugriff auf das übergeordnete Fenster speichert das Objekt eines untergeordneten Fensters einen Zeiger auf das Objekt im Objektfeld *Parent*. Auf seine eigenen untergeordneten Fenster greift es über eine verknüpfte Liste zu, die in seinem *ChildList*-Feld gespeichert ist.

Die Verwaltung von *ChildList* erfolgt automatisch. Das untergeordnete Fenster, auf das *ChildList* zeigt, ist das jeweils zuletzt erzeugte untergeordnete Fenster.

Wenn Sie einen neuen Objekttyp für ein Schnittstellenelement definieren, der über untergeordnete Fenster verfügt, müssen Sie den *Init*-Konstruktor so definieren, daß er auch die untergeordneten Fensterobjekte konstruiert.

Das Schnittstellenelement des übergeordneten Fensters wird erzeugt, wenn die Anwendung dessen *Create*-Methode aufruft. Bei erfolgreicher Ausführung ruft *Create* die *SetupWindow*-Methode auf. Die von allen Schnittstellenobjekten geerbte *SetupWindow*-Methode iteriert über jedes untergeordnete Fenster in *ChildList* und ruft per Voreinstellung dessen *Create*-Methode auf, außer, es handelt sich um ein Dialogfenster. Es ist oft nützlich, die *SetupWindow*-Methode des übergeordneten Fensters umzudefinieren, um nach der Erzeugung der untergeordneten Fenster einige Einstellungen vornehmen zu können. Das Füllen einer Liste ist ein gutes Beispiel hierfür. In diesem Fall muß die erste Zeile der neu definierten *SetupWindow*-Methode einen Aufruf der geerbten *SetupWindow*-Methode enthalten.

Wie der Aufruf der *Create*-Methode des übergeordneten Fensters zu einem Aufruf der *Create*-Methoden seiner untergeordneten Fenster führt, erzeugt auch der Aufruf des *Done*-Destruktors des übergeordneten Fensters einen Aufruf der Destruktoren seiner untergeordneten Fenster. Daraus folgt, daß Ihr Programm *Create* oder *Done* für untergeordnete Fenster nicht explizit aufzurufen braucht. Dies gilt auch für die *CanClose*-Methode, die nur dann *True* ergibt, wenn *CanClose* auch für die untergeordneten Fenster aufgerufen wurde.

Soll ein untergeordnetes Fenster (zum Beispiel ein Pop-up-Fenster) explizit vom automatischen Erstellungs- und Anzeige-Mechanismus ausgenommen werden, muß nach der Konstruktion des Objekts die Methode *DisableAutoCreate* aufgerufen werden.

Um ein untergeordnetes Fensters explizit zum automatischen Erstellungs- und Anzeige-Mechanismus hinzuzufügen (etwa ein Dialogfenster, das normalerweise nicht angezeigt würde), muß nach der Konstruktion des Objekts die Methode *EnableAutoCreate* aufgerufen werden.

Iterieren über untergeordnete Fenster

Manchmal werden Methoden benötigt, die über jedes untergeordnete Fenster iterieren und dabei eine Funktion ausführen.

Um zum Beispiel alle untergeordneten Auswahlfelder eines Fensters zu markieren, wird die Methode *TWindowsObject.ForEach* folgendermaßen eingesetzt:

```
procedure MyWindow.CheckAllBoxes;

  procedure CheckTheBox(ABox: PWindowsObject); far;
  begin
    PCheckBox(ABox)^.Check;
  end;

begin
  ForEach(@CheckAllBoxes);
end;
```

ForEach (und die verwandten Methoden *FirstThat* und *LastThat*) werden auf ähnliche Weise verwendet wie die gleichnamigen Methoden von *TCollection*. Zwar benutzt ObjectWindows zur Verwaltung von untergeordneten Fenstern keine Kollektion, die Iterator-Methoden arbeiten aber auf dieselbe Art und Weise.

Sie können auch Methoden schreiben, die über die Liste der untergeordneten Fenster iterieren und ein bestimmtes Fenster heraussuchen. Um beispielsweise aus vielen untergeordneten Auswahlfeldern das erste markierte zu finden, kann die Methode *TWindowsObject.FirstThat* folgendermaßen eingesetzt werden:

```
function MyWindow.GetFirstChecked: PWindowsObject;

  function IsThisOneChecked(ABox: PWindowsObject): Boolean; far;
  begin
    IsThisOneChecked := ABox^.GetCheck;
  end;

begin
  GetFirstChecked := FirstThat(@IsThisOneChecked);
end;
```

Botschafts-verarbeitung

ObjectWindows-Programme kommunizieren mit Windows über zwei Kanäle.

- In der einen Richtung steuert die Anwendung die Aktionen der Benutzerschnittstelle durch den Aufruf von Windows-Funktionen,
- in der anderen Richtung sendet Windows als Reaktion auf *Ereignisse* im Programm, etwa der Auswahl eines Menüpunktes (*wm_Command*) oder dem Drücken der linken Maustaste (*wm_LButtonDown*), Botschaften an die Anwendung.

Es existieren über 100 Standard-Windows-Botschaften, und es können weitere Botschaften definiert werden (vgl. Kapitel 7 »Überblick über ObjectWindows«). Abbildung 9.1 zeigt, wie Funktionen und Botschaften zur Kommunikation zwischen Windows und ObjectWindows-Programmen eingesetzt werden.

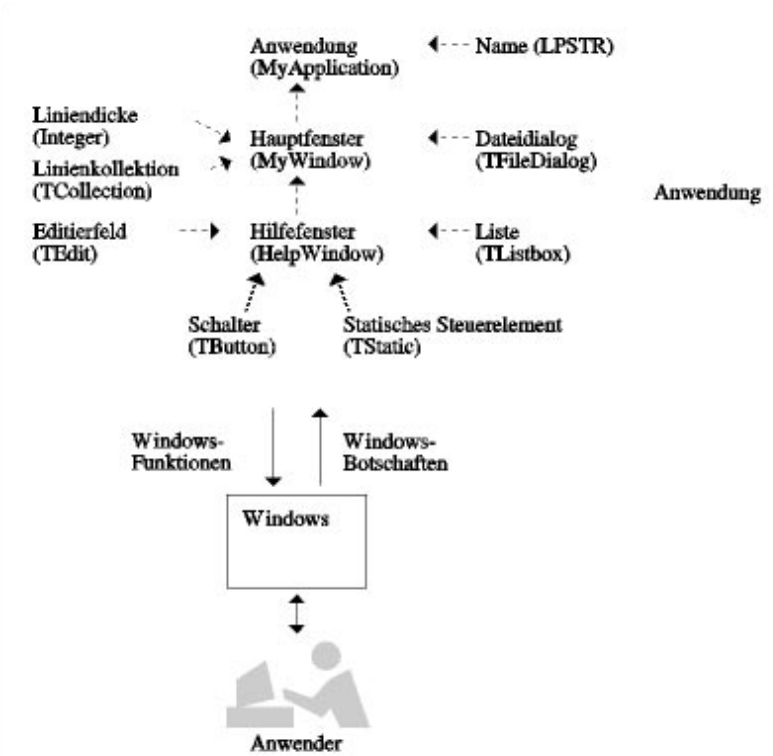


Abbildung 9.1 Die Kommunikation zwischen Windows und ObjectWindows-Programmen

Die Verarbeitung einer Windows-Anwendung erfolgt durch die Beantwortung der Botschaften, die sie von Windows erhält. Das anwendungsspezifische Verhalten wird durch selektives Antworten auf die Botschaften erzielt. Dabei können Windows-Funktionen aufgerufen werden, diese wiederum können neue Windows-Botschaften generieren. Eingehende Botschaften sind immer für ein spezifisches Fenster bestimmt – das Fenster, das vom aktuellen Ereignis direkt betroffen ist. Wenn beispielsweise in einem bestimmten Fenster die linke Maustaste gedrückt wird, »gehört« die erzeugte *wm_LButtonDown*-Botschaft zu diesem Fenster. ObjectWindows leitet die empfangenen Botschaften zu den Antwortmethoden, die für die einzelnen Fensterobjekttypen definiert wurden.

Das Zusammenspiel von Windows-Botschaften und Objektmethoden bildet das Grundgerüst einer ObjectWindows-Anwendung. Die Hauptaufgabe einer ObjectWindows-Anwendung besteht demnach in der Beantwortung von Windows-Ereignissen. Und hierin besteht auch der große Unterschied zur traditionellen programmgesteuerten DOS-Anwendung. Windows-Anwendungen erfordern eine auf Botschaften basierende, ereignisgesteuerte Programmierung. Diese Art der Programmierung erfordert erhöhte Disziplin, liefert dafür aber effektive und Multitasking-fähige Anwendungen.

Antworten auf Botschaften

Damit Fensterobjekte auf eingehende Windows-Botschaften antworten, muß für jede Botschaft (die beantwortet werden soll) eine eigene Methode definiert werden. Derartige Methoden werden als *Botschaftsantwort-Methoden* bezeichnet. Ihre Erstellung wird durch eine Erweiterung der virtuellen Methodendefinition erleichtert. Um eine Methode (es handelt sich dabei immer um eine Prozedur) als Botschaftsantwort-Methode zu kennzeichnen, wird die Summe aus *wm_First* und dem Botschaftsnamen an das Ende des Kopfs der virtuellen Methodendefinition angehängt:

```
type
  TMyWindow = object(TWindow)
    ...
    procedure WMRButtonDown(var Msg: TMessage); virtual wm_First +
      wm_RButtonDown;
  ...
```

In diesem Beispiel wird die *WMRButtonDown*-Methode aufgerufen, wenn ein *TMyWindow*-Objekt eine *wm_RButtonDown*-Botschaft empfängt.

Methoden- und Botschaftsnamen müssen sich zwar nicht ähneln, die Lesbarkeit des Programms läßt sich aber dadurch verbessern. ObjectWindows verwendet diese Konvention bei den Typen der Schnittstellenobjekte. *Msg* ist ein erforderliches Variablenargument vom Record-Typ, der Felder zur Aufnahme der in der Windows-Botschaft enthaltenen Word- und Longint-Parameter (*wParam* bzw. *lParam*) und ein Feld zum Speichern des Ergebnisses (*Result*) definiert.

wParam und *lParam* enthalten Informationen über die Aktion, durch die die Botschaft ausgelöst wurde. Zum Beispiel übergibt eine *wm_LButtonDown*-Botschaft die X/Y-Koordinaten des angeklickten Punktes in *lParam*. Häufig ist in jedem Parameter eine zweiteilige Information enthalten. Wird *wm_LButtonDown* übergeben, enthält das niederwertige Word von *lParam* die X-Koordinate und das höherwertige Word die Y-Koordinate.

TMessage enthält außerdem zusätzliche variante Felder für die Aufnahme der höher- und niederwertigen Teile von *wParam*, *lParam* und *Result*. *wParamLo* und *wParamHi* sind *Byte*-Felder zur Aufnahme des höher- und niederwertigen Bytes des Word *wParam*. *lParamLo* und *lParamHi* sind Word-Felder, in denen die höher- und niederwertigen Wörter des Longint-Parameters *lParam* gespeichert werden. *ResultLo* und *ResultHi* sind Word-Felder zur Aufnahme der höher- und niederwertigen Wörter des Longint *Result*.

Die Felder *Lo* und *Hi* sind keine Kopien der ursprünglichen Daten, eine Modifizierung von *lParamLo* hat somit eine Änderung von *lParam* zur Folge.

In Ihren Programmen werden Sie die Felder *wParam*, *lParamLo* und *lParamHi* häufig verwenden. Sie sind alle vom Typ Word. *Msg* wird als Variablenargument übergeben, da einige Windows-Botschaften eine Antwort erfordern, die von den Methoden im *Result*-Feld eines *TMessage*-Record gespeichert werden.

Windows-Botschaften, auf die Ihre Objekte nicht antworten sollen, können Sie ignorieren. Sie brauchen in einem derartigen Fall keine Botschaftsantwort-

Methode zu definieren. Um die entsprechenden Botschaften kümmert sich die Methode *DefWndProc*, die das Fenster von *TWindowsObject* erbt.

Für die Beantwortung einiger Botschaften, die im Zusammenhang mit Menüs und untergeordneten Fenstern häufig vorkommen, besteht die Möglichkeit, Kürzel zu verwenden. Näheres erfahren Sie in den folgenden Abschnitten.

Befehlsabhängige Botschaften und Botschaften untergeordneter Fenster

Windows erzeugt eine befehlsabhängige Botschaft, wenn der Anwender einen Menüpunkt auswählt oder einen Accelerator eingibt. Aktiviert er dagegen ein Steuerelement, zum Beispiel einen Schalter oder eine Liste, generiert Windows eine Botschaft, die von der ID des untergeordneten Fensters abhängig ist. Die meisten derartigen Aktionen erzeugen eine *wm_Command*-Botschaft.

ObjectWindows antwortet auf *wm_Command*-Botschaften durch die automatische Erzeugung eines spezialisierteren, virtuellen Methodenaufrufs, der auf dem Inhalt der jeweiligen Botschaft basiert. Sie können sich daher das Schreiben umfangreicher case-Anweisungen sparen. Bei der Definition von Antwortmethoden für Botschaften, die von Befehlen oder untergeordneten Fenstern abhängig sind, kann eine Erweiterung der Methodendefinition – ähnlich der bereits beschriebenen – benutzt werden.

Verarbeitung befehlsabhängiger Botschaften

Verwenden Sie bei der Definition einer Antwortmethode für befehlsabhängige Botschaften (Menü- oder Accelerator-Ereignis) einen Bezeichner, der aus der Summe zweier Konstanten besteht: Aus der in ObjectWindows definierten Konstanten *cm_First* und aus der Menü- oder Accelerator-ID, die in der Menü- bzw. Accelerator-Ressource definiert ist und in der *wm_Command*-Botschaft übergeben wird.

Angenommen, ein Fenster (ein *TFileWindow*-Objekt) enthält die drei Menüpunkte File/New, File/Open und File/Save. Die in der Ressourcendatei definierten Menü-IDs lauten 101, 102 und 103. Die Objektdefinition von *TFileWindow* könnte dann folgendermaßen aussehen:

```
TFileWindow = object(TWindow)
  ...
  procedure CMFileNew(var Msg: TMessage); virtual cm_First + 101;
  procedure CMFileOpen(var Msg: TMessage); virtual cm_First + 102;
  procedure CMFileSave(var Msg: TMessage); virtual cm_First + 103;
end;
```

Zur Verbesserung der Lesbarkeit können für jede ID Konstanten definiert werden.

Wird zum Beispiel *cm_FileNew* als 101 definiert, sieht der Methodenkopf folgendermaßen aus:

```
procedure CMFileNew(var Msg: TMessage); virtual cm_First +
  cm_FileNew;
```

Ein Fenster mit einem Menü muß aber nicht immer eine Antwort auf befehlsabhängige Botschaften definieren. Diese Aufgabe kann auch ein untergeordnetes Fenster (auch ein Steuerelement) übernehmen. Das untergeordnete Fenster, das zum Zeitpunkt der Menüauswahl bzw. der Accelerator-Eingabe den Fokus besitzt, hat die erste Gelegenheit zur Beantwortung einer befehlsabhängigen Botschaft. Wenn es keine Antwortmethode definiert hat, wird die Botschaft an das übergeordnete Fenster weitergegeben. Wenn aber das untergeordnete Fenster die Botschaft bearbeitet hat und diese auch vom übergeordneten Fenster empfangen werden soll, muß sie explizit durch einen Aufruf der Routine *DefCommandProc* weitergegeben werden.

Verarbeitung von Botschaften untergeordneter Fenster

Beeinflußt der Anwender etwa durch das Anklicken eines Schalters oder die Eingabe in ein Editierfeld ein Steuerelement (ein untergeordnetes Fenster), erhält das Objekt des übergeordneten Fensters eine von der ID des untergeordneten Fensters abhängige Botschaft. Bei der Definition einer Methode zur Beantwortung einer derartigen Botschaft muß ein Bezeichner verwendet werden, der sich aus der Summe zweier Konstanten zusammensetzt:

- Aus der in ObjectWindows definierten Konstanten *id_First* und
- aus der ID des untergeordneten Fensters.

Die letztere kann in einer Dialogressource oder bei der Konstruktion eines Steuerelementobjekts definiert sein.

Enthält beispielsweise ein Fenster eine Liste, kann zur Beantwortung der von der Liste generierten Botschaften eine Methode mit dem Namen *IDListBox* definiert werden:

```
TListBoxWindow = object(TWindow)
  ...
  procedure IDListBox(var Msg: TMessage); virtual id_First +
   id_ListBox;
end;
```

Hierbei ist *id_ListBox* eine Konstante, die die Steuerelement-ID repräsentiert. Als IDs von untergeordneten Fenstern sind nur positive ganze Zahlen zulässig, die kleiner als 4094 sind.

Alternativ zur Botschaftsbeantwortung durch das übergeordnete Fenster kann die Antwort auch direkt durch das Steuerelement erfolgen. Näheres hierzu finden Sie im Abschnitt »Antwort auf Benachrichtigungsbotschaften der Steuerelemente« in Kapitel 12 »Steuerelementobjekte.«

Wie bei den befehlsabhängigen Botschaften können auch die Botschaften untergeordneter Fenster über eine Kette von Schnittstellenelementen weitergegeben werden, wobei beim betroffenen untergeordneten Fenster begonnen wird (die Botschaft wird dadurch zur Benachrichtigungsbotschaft). Darauf erfolgt die Übergabe der Botschaft an das übergeordnete Fenster. Eine Antwortmethode des untergeordneten Fensters kann verhindern, daß die Botschaft das übergeordnete Fenster erreicht, indem sie das *Result*-Feld des *TMessage*-Recordparameters auf 1 setzt.

Botschaftsverarbeitung durch Standardantworten

Dieser Abschnitt befaßt sich mit Windows-Botschaften, die direkt (beginnend mit *wm_First*) abgefangen werden, die aber nicht von Befehlen oder von der ID untergeordneter Fenster abhängig und keine Benachrichtigungsbotschaften sind. Eine typische Windows-Anwendung empfängt von Windows viele Botschaften, die der Programmierer nicht beantworten will. So kann zum Beispiel bereits das Anklicken eines Fensters 10 oder mehr Botschaften auslösen. Für die Erstellung Ihrer Botschaftsantwort-Methoden sollten Sie daher auch wissen, was passiert, wenn eine Anwendung Meldungen ignoriert.

Wenn Windows einer Anwendung eine Botschaft übergibt, muß es wissen, ob sie abgefangen und bearbeitet wurde. Andernfalls setzt Windows die Standardantworten für ignorierte Botschaften ein. Gibt der Anwender beispielsweise Text in ein Editierfeld ein, antwortet Windows automatisch mit der Anzeige von Zeichen und (falls erforderlich) mit dem Rollen des Textes. Der Aufruf dieser Standardantworten erfolgt durch die voreingestellte Fensterprozedur des Schnittstellenobjekts, der *DefWndProc*-Methode.

Wenn Sie eine Windows-Botschaft ignorieren (d.h. keine Antwortmethode definieren), ruft ObjectWindows automatisch die voreingestellte Fensterprozedur des Schnittstellenobjekts auf. Der Aufruf von *DefWndProc* kann aber auch dann erforderlich sein, wenn Sie die Botschaft durch die Definition einer Antwortmethode abfangen.

Zum Beispiel fängt das folgende Programmfragment die Botschaft *wm_Char* ab, die nach einer Eingabe des Anwenders in ein Editierfeld an das entsprechende Editierfeld-Objekt gesendet wird. Die (von *TEdit* abgeleitete) Antwortmethode für *wm_Char* sieht folgendermaßen aus:

```
procedure TMyEdit.wmChar(var Msg: TMessage);
var
  TheText: array[0..255] of Char;
begin
  MessageBeep(0);
end;
```

Diese Methode antwortet auf die Eingabe des Anwenders mit einem Signalton. Die Definition der Antwortmethode verhindert aber die Standardantwort der voreingestellten Fensterprozedur. Das hat zur Folge, daß zwar der Signalton ertönt, der eingegebene Text aber nicht im Editierfeld erscheint.

Damit der Signalton ertönt *und* die Standardantwort ausgelöst wird, muß die *DefWndProc*-Methode explizit aufgerufen werden:

```
procedure TMyEdit.wmChar(var Msg: TMessage);
var
  TheText: array[0..255] of Char;
begin
  DefWndProc(Msg);
  MessageBeep(0);
end;
```

Diese Methode sorgt dafür, daß das eingegebene Zeichen angezeigt wird und anschließend der Signalton ertönt.

DefWndProc muß in folgenden Fällen aufgerufen werden:

- Wenn eine Botschaft zwar abgefangen werden soll (beispielsweise zu Benachrichtigungszwecken), daraus aber keine Aktion resultieren soll.
- Wenn eine Botschaft eine andere Aktion als die Standardantwort auslösen soll, die Standardantwort aber trotzdem benötigt wird.
- Wenn das Abfangen einer Botschaft die Benutzung des zugehörigen Schnittstellenelements unmöglich macht.

10. Fensterobjekte

In diesem Kapitel erfahren Sie, wie die Fenster einer Anwendung erstellt, angezeigt und ausgefüllt werden. Der ObjectWindows-Objekttyp *TWindow* dient als Vorlage für die Definition des Verhaltens und der Eigenschaften des Programmhauptfensters und seiner weiteren Pop-up-Fenster (falls vorhanden).

Während Instanzen von *TWindows* generische Fenster sind, definiert der Objekttyp die Protokolle für das Öffnen und Schließen von Fenstern sowohl für untergeordnete Fenster als auch für die Bearbeitung von Befehlsbotschaften.

Von *TWindow* stammen drei Typen ab: *TMDIWindow*, *TControl* und *TEditWindow*. Die MDI-Typen werden in ObjectWindows in denjenigen Anwendungen eingesetzt, die der Multidokumentschnittstelle entsprechen (MDI steht als Abkürzung für »Multiple Documents Interface«). Eine Erläuterung der MDI und der dazugehörigen Typen finden Sie in Kapitel 13 »MDI-Objekte«. *TControl* definiert Steuerelemente wie Schalter und Listen und wird in Kapitel 12 »Steuerelementobjekte« erläutert. Weitaus am häufigsten werden Sie jedoch Fenstertypen anlegen, die von *TWindow* abstammen.

In diesem Kapitel werden die Typen *TWindow* und *TEditWindow* erläutert, außerdem finden Sie ein Beispiel für die Registrierung neuer Windows-Klassen.

Der Typ TWindow

Eine ObjectWindows-Anwendung muß zumindest über ein Hauptfenster verfügen, das beim Start der Anwendung auf dem Bildschirm angezeigt wird. Das *TestApp*-Programm in Kapitel 8 »Anwendungsobjekte« ist ein Beispiel für ein ObjectWindows-Programm, das dieser Mindestforderung entspricht. Normalerweise ist das Hauptfenster eines ObjectWindows-Programms eine Objektinstanz vom Typ *TWindow* bzw. häufiger ein Nachkomme eines vom Programm definierten Typs. Viele Anwendungen haben weitere Fenster, die üblicherweise untergeordnete Fenster des Hauptfensters sind. Diese zusätzlichen Fenster sind ebenfalls Objektinstanzen des Typs *TWindow* oder (normalerweise) ein Nachkomme davon. So kann ein Zeichenprogramm beispielsweise für sein Hauptfenster den Typ *TPaintWindow* und für ein anderes Fenster den Typ *TZoomWindow* definieren, der eine vergrößerte Darstellung des Gezeichneten anzeigt. In diesem Fall stammen beide Typen von *TWindow* ab. In einer effizient gestalteten

Anwendung wäre *ZoomWindow* eine spezielle Version von *TPaintWindow* und als direkter Nachkomme dieses Typs (und als indirekter Nachkomme von *TWindow*) implementiert.

Initialisieren und Erstellen von Fensterobjekten

Wie Dialog- und Steuerelementobjekte sind Fensterobjekte Schnittstellenobjekte, die mit visuellen Schnittstellenelementen verbunden sind. Genau gesagt, repräsentieren sie Fensterelemente, die durch ein Handle verknüpft sind, das in dem von *TWindowsObject* geerbten *HWindow*-Feld gespeichert ist. Da ein Fensterobjekt aus zwei Teilen besteht, wird dieses Fenster in zwei Schritten erstellt und zwar in folgender Reihenfolge: Initialisierung des Objekts und Erstellen des visuellen Elements.

Initialisieren des Fensterobjekts

Eine typische Windows-Anwendung verfügt über viele unterschiedliche Fenstertypen: überlappende oder Pop-up-, gerahmte, rollbare oder betitelte Fenster und vieles mehr. Diese Stilattribute werden, ebenso wie Fenstertitel und -menü, während der Initialisierung eines Fensterobjekts bestimmt und während des Erstellens eines Fensterelements verwendet.

Die Erstellungsattribute eines Fensterobjekts, wie Stil, Titel und Menü, werden in dem von ObjectWindows definierten Recordtyp *TWindowAttr* gespeichert. Jedes Fensterobjekt speichert einen *TWindowAttr*-Record in seinem *Attr*-Objektfeld. Ein derartiger Record setzt sich aus folgenden Feldern zusammen:

Tabelle 10.1 Fensterattribut-Felder

Feld	Verwendung
Title	Ein PChar, das den Titel-String enthält.
Style	Ein Longint, der die kombinierte Stilkonstante enthält.
Menu	Ein Handle auf eine Menüressource (*HMenu*).
X	Ein Integer, der die horizontale Startposition des Fensters enthält. Dabei handelt es sich um die horizontale Bildschirmkoordinate der linken oberen Fensterecke.
Y	Ein Integer, der die vertikale Startposition des Fensters enthält. Dabei handelt es sich um die vertikale Bildschirmkoordinate der linken oberen Fensterecke.
W	Ein Integer, der die anfängliche Fensterbreite in Bildschirmkoordinaten angibt.
H	Ein Integer, der die anfängliche Fensterhöhe in Bildschirmkoordinaten angibt.

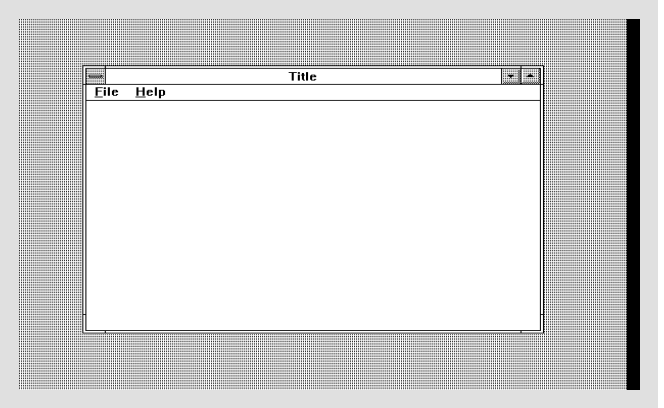


Abbildung 10.1
Fensterattribute

Die Initialisierung des Fensters ist der Vorgang, bei dem das ObjectWindows-Fensterobjekt durch Aufruf des Konstruktors *Init* erstellt wird.

```
Window1 := New(PWindow, Init(nil, 'Title of Window 1'));
Window2 := New(PNewWindowType, Init(nil, 'Title of Window 2');
```

Init legt das neue Fensterobjekt an und besetzt das *Title*-Feld von *Attr* mit dem übergebenen PChar-Argument. Ist das Fenster das Hauptfenster der Anwendung, wird *Style* standardmäßig auf *ws_OverlappedWindow* or *ws_Visible* gesetzt, ansonsten auf *ws_Visible*. *X*, *Y*, *W* und *H* werden auf *cw_UseDefault* gesetzt, wodurch ein überlappendes Fenster normaler Größe entsteht. Auf diese Weise wird standardmäßig die Größe eines neuen Hauptfensters festgelegt. Das erste Argument im *Init*-Aufruf ist das Fensterobjekt des übergeordneten Fensters. Es ist nil, wenn das Fenster das Hauptfenster ist.

Wenn Sie von *TWindow* neue Fenstertypen ableiten, definieren Sie normalerweise einen neuen *Init*-Konstruktor, vor allem dann, wenn Sie ein nicht standardmäßiges Erstellungsattribut haben wollen. Möchten Sie *Init* nicht neu definieren, können Sie die Attribute für ein Fensterobjekt neu bestimmen, indem Sie das *Attr*-Feld des Objekts unmittelbar nach dem Aufruf von *Init* ändern. Bei der Neudefinition von *Init* müssen Sie sicherstellen, daß *Init* als erstes das geerbte *TWindow.Init* aufruft, das die Standardattribute setzt. Danach können Sie die gewünschten Attribute ändern. So würde ein gängiger Fenstertyp zum Beispiel ein *Init* definieren, das das *Menu*-Attribut setzt:

```
constructor AWindowType.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  AChildWindow := New(PChildWindowType, Init(@Self, 'Child Title'));
  List1 := New(PListBox, Init(@Self, id_ListBox, 201, 20, 20,
    180, 80));
  ...
end;
```

Beachten Sie, daß im obigen Beispiel der Fenster-Konstruktor *Init* für die Konstruktion der untergeordneten Fenster zuständig ist, wie zum Beispiel für Pop-up-Fenster und Listen. Umgekehrt kann ein untergeordneter Fenstertyp seine Attribute in seinem eigenen *Init*-Konstruktor neu setzen:

```
constructor ChildWindowType.Init(AParent: PWindowsObject;
  ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  with Attr do
  begin
    Style := Style or ws_PopupWindow or ws_Caption;
    X := 100; Y := 100; W := 300; H := 300;
  end;
end;
```

Das Attribut *Style* kann eine Fensterstilkonstante oder eine Kombination mehrerer Stilkonstanten sein, wie beispielsweise

```
ws_OverlappedWindow
ws_PopupWindow
ws_ChildWindow
ws_Caption
ws_Border
ws_VScroll
```

Diese Liste enthält einige gebräuchliche Stilkonstanten. Eine Zusammenfassung aller Konstanten finden Sie in Kapitel 1 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch*. Wollen Sie keinen abgeleiteten Fenstertyp definieren, können Sie auch zuerst das Fensterobjekt konstruieren und dann dessen Attribute mit dem *Init*-Konstruktor des übergeordneten Fensters setzen:

```
constructor AWindowType.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar (100));
  AChildWindow := New(PChildWindowType, Init(@Self, 'Child Title'));
  with AChildWindow^.Attr do

  begin
    Style := Style or ws_PopupWindow or ws_Caption;
    X := 100; Y := 100; W := 300; H := 300;
  end;
end;
```

Erstellen von Fensterelementen

Das Erstellen von Fensterelementen ist der Aufbauprozeß von visuellen Elementen, die mit einem Fensterobjekt verknüpft sind. Dieser Aufbau erfolgt

durch den Aufruf der Methode *MakeWindow* des Anwendungsobjekts, dem das Fensterobjekt als Parameter übergeben wird:

```
if Application^.MakeWindow(AWindow) <> nil then
  { Element erstellt }
else { Element nicht erstellt }
```

MakeWindow ruft zwei wichtige Methoden auf. Die erste trägt den Namen *ValidWindow* und überprüft, ob das Fensterobjekt erfolgreich konstruiert wurde. Wenn der Konstruktor aus irgendeinem Grund nicht erfolgreich war, liefert *MakeWindow* nil zurück. War der Konstruktor erfolgreich, setzt *MakeWindow* seine Arbeit fort und ruft die *Create*-Methode auf, die Windows dazu veranlaßt, ein visuelles Element zu erzeugen. Wenn *Create* nicht erfolgreich ist, liefert *MakeWindow* nil zurück, andernfalls wird ein Zeiger auf das Fensterobjekt geliefert.

Obwohl *Create* die Methode ist, die das Fensterelement tatsächlich erstellt, werden Sie *Create* normalerweise nicht explizit aufrufen. Das Hauptfenster der Anwendung wird beim Start der Anwendung durch *TApplication.InitInstance* erzeugt. Alle anderen Fenster in der Anwendung sind untergeordnete Fenster, die direkt oder indirekt vom Hauptfenster abstammen. Untergeordnete Fenster werden normalerweise in der *SetupWindow*-Methode ihrer übergeordneten Fensterobjekte oder dynamisch mit *MakeWindow* erstellt.

Zusammenfassung

Wenn Sie einen neuen Fensterobjekttyp entwerfen, können Sie *Init* so neudefinieren, daß die Fensterattribute gesetzt und die untergeordneten Fensterobjekte (falls vorhanden) konstruiert werden. (Ein Fensterattribut kann auch direkt gesetzt werden, zum Beispiel im *Init* des übergeordneten Fensters.) *MakeWindow* baut ein visuelles Element für das Fensterobjekt auf. Zu diesem Zweck wird *Create* aufgerufen, das wiederum die Methode *InitWindow* des Fensterobjekts aufruft, die die untergeordneten Fenster erstellt.

Im allgemeinen rufen übergeordnete Fenster *Init* und *MakeWindow* für ihre untergeordneten Fenster auf. Die Attribute eines Fensterobjekts werden normalerweise von den eigenen bzw. von den Methoden des übergeordneten Fensterobjekts gesetzt. Da das Hauptfenster einer Anwendung kein übergeordnetes Fenster hat, wird es vom Anwendungsobjekt beim Start der Anwendung konstruiert und erzeugt.

Fensterklassen-Registrierung

Wie Sie gesehen haben, können bei der Initialisierung von Fensterobjekten die Attribute des Fensters (wie Stil, Plazierung und Menü) durch Ausfüllen des Objektfeldes *Attr* bestimmt werden. Diese Attribute werden beim Erstellen

des entsprechenden Fensterelements verwendet und daher *Erstellungsattribute* genannt.

Es gibt jedoch noch einige weitere Attribute, wie Hintergrundfarbe, Darstellungssymbol und Mauszeiger. Wollen Sie sich ein Beispiel hierfür ansehen, so lassen Sie gleichzeitig unterschiedliche Anwendungen in Windows ablaufen. Wenn Sie den Mauszeiger von einer Anwendung in eine andere bewegen, wechselt der Zeiger sein Aussehen: aus dem (regulären) Pfeil wird ein dickes Kreuz bei Kalkulationsprogrammen und ein dünnes Kreuz bei Grafikprogrammen.

Diese Attribute sind mit dem Fensterobjekttyp enger verknüpft und können während der Programmoperation nicht geändert werden, wie dies bei den Erstellungsattributen der Fall ist. Zur Unterscheidung nennen wir diese inhärenten Attribute *Registrierungsattribute*.

Mit jedem Fensterobjekt ist eine Liste mit Registrierungsattributen verknüpft, die sogenannte *Fensterklasse*. Die Liste der Registrierungsattribute ähnelt der Liste der Erstellungsattribute, die als *Attr*-Record in einem Objektfeld eines Fensterobjekts gespeichert ist. Die Registrierungsattribute sind jedoch in einem Record namens *TWndClass* abgelegt, der von Windows definiert und verwaltet wird.

Den Vorgang, bei dem eine Fensterklasse mit einem Fensterobjekttyp verknüpft wird, nennen wir *Registrieren* einer Fensterklasse. ObjectWindows führt diesen Registrierungsprozeß selbständig durch. Aus diesem Grund müssen Sie sich um die Registrierung nicht kümmern, außer Sie wollen eines der voreingestellten Fenstercharakteristika ändern.

Möchten Sie ein Registrierungsattribut modifizieren, beispielsweise den Zeiger oder das Darstellungssymbol, müssen Sie eine neue Fensterklasse definieren, indem Sie zwei Methoden schreiben: *GetClassName* und *GetWindowClass*. *GetClassName* ist eine Funktion, die einfach den Namen der Fensterklasse (PChar) zurückliefert. *TWindow* definiert eine *GetClassName*-Methode, die 'TurboWindow', den Namen der voreingestellten Fensterklasse, zurückliefert:

```
function TWindow.GetClassName: PChar;

begin
  GetClassName := 'TurboWindow';
end;
```

Zur Definition eines Fensterobjekttyps namens *IBeamWindow*, der anstelle des Standardpfeils einen I-Balkenzeiger verwendet, definieren Sie die geerbte *GetClassName*-Methode folgendermaßen neu:

```
function IBeamWindow.GetClassName: PChar;

begin
  GetClassName := 'IBeamWindow';
end;
```

Der Name der Klasse kann – muß aber nicht – derselbe sein wie der Name des Objekttyps.

Die Methode *GetWindowClass* hat einen *TWndClass*-Record als Variablenparameter und füllt dessen Felder mit den neuen Registrierungsattributen aus.

Ähnlich wie beim Definieren einer neuen *Init*-Methode zur Festlegung der Attribute eines Fensters sollten Sie als erstes *TWindow.GetWindowClass* aufrufen. Dann setzen Sie die Felder, die geändert werden sollen. Das Feld *hCursor* beinhaltet zum Beispiel ein Handle auf eine Zeigerressource. Definieren Sie für *IBeamWindow* die folgende *GetWindowClass*-Methode:

```
procedure IBeamWindow.GetWindowClass(var AWndClass: TWndClass);

begin
  TWindow.GetWindowClass(AWndClass);
  AWndClass.hCursor := LoadCursor (0, idc_IBeam);
end;
```

idc_Beam ist eine Konstante, die einen der vordefinierten Windows-Zeiger repräsentiert. In Abbildung 10.2 sehen Sie eine Anwendung, die das *IBeamWindow*-Objekt einsetzt.

Abbildung 10.2 Ein Programm, das die Klasse IBeamWindow benutzt.

Außer Fenstern benötigen auch komplexe Dialogfenster (*TDlgWindow*) registrierte Fensterklassen (vergleichen Sie hierzu Kapitel 5 »Konstruktion eines Dialogs«). Dagegen erfordern einfache Dialogfenster und Steuerelemente keine Fensterklassen.

Registrierungsattribute

Der Typ *TWindow* definiert eine Fensterklasse 'TurboWindow', die ein leeres Symbol, einen Pfeilzeiger und eine Standardfensterfarbe aufweist. Wenn Sie Ableitungen von diesen Charakteristika anlegen möchten, müssen Sie die Felder eines *TWndClass*-Records mit anderen Daten in einer *GetWindowClass*-Methode belegen.

In der folgenden Tabelle 10.2 sehen Sie die Charakteristikafelder eines *TWndClass*-Records und ihre von *TWindow.GetWindowClass* definierten Standardwerte:

Tabelle 10.2 Registrierungsattribute

Charakteristika	Feld	Standardwert
Klassenstil	*style*	*cs_HRedraw* or *cs_VRedraw*
Symbol	*hIcon*	*LoadIcon(0, idi_Application)*
Zeiger	*hCursor*	*LoadCursor(0, idc_Arrow)*
Hintergrundfarbe	*hbrBackground*	*HBrush(Color_Window + 1)*
Standardmenü	*lpszMenuName*	nil

Das Feld für den Klassenstil

Das Feld *style* unterscheidet sich vom Fenster-Stilattribut (*ws*), das Sie während der Fensterinitialisierung festlegen, denn es legt Verhalten und Eigenschaften fest, die der Funktionsweise eines Fensters im Gegensatz zu dessen visuellen Erscheinungsformen inhärent sind. Das Feld kann mit einer Klassenstilkonstante oder einer Kombination von Konstanten (*cs*) ausgefüllt werden. *cs_HRedraw* baut beispielsweise das gesamte Fenster neu auf, wenn sich die horizontale Größe ändert, *cs_NoClose* verhindert die *Close*-Option im Systemmenü, und *cs_ParentDC* weist dem Fenster den Bildschirmkontext des übergeordneten Fensters zu. Eine vollständige Liste aller *cs*-Konstanten finden Sie in Kapitel 1 »Windows-Stile und -Konstanten« im *Windows-Referenzhandbuch*.

Die Klasse *TurboWindow* definiert eine Fensterklasse mit folgendem Stil:

```
AWndClass.style := cs_HRedraw or cs_VRedraw;
```

Das Feld für das Symbol

Das Feld *hIcon* enthält ein Handle auf ein Symbol, das zur Repräsentation eines Fensters in seinem minimalisierten Status dient. Normalerweise werden Sie eine Symbolressource definieren, die das Hauptfenster der Anwendung darstellt. Per Voreinstellung verwendet die *TurboWindow*-Klasse das vordefinierte Symbol *idi_Application*, das als leeres Rechteck angezeigt wird.

Das Feld für den Zeiger

Das Feld *hCursor* enthält ein Handle auf einen Zeiger, der der Repräsentation des Mauszeigers dient, wenn er über dem Fenster steht. Die Klasse *TurboWindow* verwendet den Standard-Windows-Pfeil *idc_Arrow*. Außerdem stehen folgende vordefinierte Zeiger zur Verfügung:

```
idc_IBeam
idc_Wait
idc_Cross
```

Das Feld für die Hintergrundfarbe

Das Feld *hbrBackground* legt die Hintergrundfarbe für das Fenster fest. Für die meisten Anwendungen ist die voreingestellte Standardfarbe ausreichend, die der Anwender in der Systemsteuerung bestimmen kann. Sie können jedoch eine bestimmte Farbe angeben, indem Sie in diesem Feld ein Handle auf einen physikalischen Pinsel speichern. Alternativ dazu können Sie auch einen der Windows-Farbwerte angeben, wie zum Beispiel *Color_ActiveCaption*, wobei zu jedem Farbwert 1 addiert werden muß.

Das Feld für das Standardmenü

Das Feld *lpszMenuName* zeigt auf den Namen einer Menüressource, die als voreingestelltes Menü für die Klasse fungieren soll. Wenn Sie beispielsweise einen *EditWindow*-Typ definiert haben, der immer über ein Standard-Editiermenü verfügen soll, so könnten Sie das betreffende Menü hier festlegen. Dadurch ist es nicht mehr erforderlich, das Menü in der *Init*-Methode zu bestimmen. Hat die Menüressource eine ID von 101, können Sie das Feld folgendermaßen belegen:

```
AWndClass.lpszMenuName := LoadMenu(HInstance, PChar(101));
```

Editier- und Dateifenster

ObjectWindows stellt zwei Nachkommen von *TWindow* zur Verfügung, die speziell zum Editieren von Fenstern geeignet sind. Der Objekttyp *TEditWindow* besitzt einen einfachen Texteditor, der aber keine Datei lesen oder in eine Datei schreiben kann. Der Typ *TFileWindow*, der von *TEditWindow* abstammt, verfügt über einen Texteditor, der die vorhin genannten Fähigkeiten hat. Diese Objekttypen können direkt als Standardkomponenten in Ihrer Anwendung eingesetzt werden. Sie können von diesen Typen Ihre eigenen Typen ableiten, um spezialisierte Editoren zu erzeugen. Programme mit Editier- oder Dateifenstern müssen die Unit *StdWnds* in der uses-Anweisung aufführen.

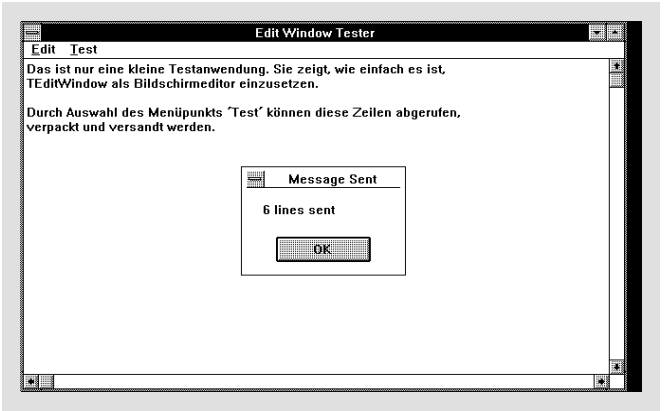


Abbildung 10.3
Ein Editierfenster

Editierfenster

Ein Editierfenster ist ein Fenster, das in seiner ganzen Größe ein Editierfeld darstellt. Der *Init*-Konstruktor von *TEditWindow* initialisiert das *Editor*-Objektfeld von *TEditWindow* so, daß es auf ein Editierfeld zeigt. Die *SetupWindow*-Methode von *TEditWindow* legt fest, daß die Größe des Editierfeldes dem

Client-Bereich des Fensters entspricht, und erzeugt das Schnittstellenelement des Editierfeldes. Die Methode *WMSize* gewährleistet, daß bei einer Größenänderung des Fensters das Editierfeld ebenfalls in seiner Größe geändert wird. Die Methode *WMSetFocus* stellt sicher, daß das Editierfeld den Fokus erhält, wenn das dazugehörige Fenster eine *wm_SetFocus*-Botschaft empfängt.

Das folgende Programm verwendet ein Editierfenster, in dem der Anwender Text für eine einfache (nicht-funktionale) elektronische Mailbox-Anwendung ändern kann (siehe Abbildung 10.3).

```
program EWndTest;
{$R EWNDTEST.RES}
uses WObjects, WinTypes, WinProcs, Strings, StdWnds;

const
  cm_SendTest = 399;

type
  TTestApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

PTestWindow = ^TTestWindow;
TTestWindow = object(TEditWindow)
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure HandleSend(var Msg: TMessage);
    virtual cm_First + cm_SendText;
end;

constructor TTestWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TEditWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, MakeIntResource(102));
end;

procedure TTestWindow.HandleSend(var Msg: TMessage);
var
  Lines: Integer;
  Text: array[0..20] of Char;
begin
  Lines := Editor^.GetNumLines;
  Str(Lines, Text);
  StrCat(@Text, 'lines sent');
  MessageBox(HWindow, @Text, 'Message Sent', mb_Ok);
end;

{ Init the main window of the TestApplication - an EditWindow }

procedure TTestApplication.InitMainWindow;
begin
  MainWindow := New(PTestWindow, Init(nil, 'Edit Window Tester'));
  HAccTable := LoadAccelerators(HInstance, MakeIntResource(100));
end;
```

```
var
  TestApp: TTestApplication;

begin
  TestApp.Init('EWndTest');
  TestApp.Run;
  TestApp.Done;
end.
```

Datei-fenster

Ein Dateifenster ist ein Editierfenster mit zusätzlichen Möglichkeiten, die das Lesen und Beschreiben von Dateien erlauben. Der *Init*-Konstruktor von *TFileWindow* übernimmt den Fenstertitel als Argument. Er initialisiert das Objektfeld *FileDialog*, um auf ein Datei-Dialogobjekt zu zeigen. *TFileWindow* verfügt über vier Methoden zur Bearbeitung von Dateien: *Open*, *Save* und *SaveAs* verwenden das Datei-Dialogfeld *TFileWindow.FileDialog*, um den Anwender zur Angabe eines Dateinamens aufzufordern (vgl. Kapitel 5 »Konstruktion eines Dialogs«), bei *New* hat der Anwender die Möglichkeit zum Abbruch, wenn das Editieren einer neuen Datei zum Verlust der Änderungen am aktuellen Text führen würde. Legen Sie ein neues Menü mit folgenden IDs an, damit der Anwender auf diese Methoden zugreifen kann:

Tabelle 10.3 Dateifenstermethoden und Menü-IDs

Methode	Menü-ID für den Aufruf
New	*cm_FileNew*
Open	*cm_FileOpen*
Save	*cm_FileSave*
SaveAs	*cm_FileSaveAs*

Sie können Dateifenster ohne Modifikation als selbständige Texteditoren verwenden. Sicher wird es aber auch vorkommen, daß Sie zur Erhöhung der Funktionalität Ihre eigenen Typen von *TFileWindow* ableiten wollen, beispielsweise wenn Sie eine Suchmöglichkeit benötigen. Bedenken Sie, daß Sie noch immer Zugriff auf das *TFileWindow.Editor*-Editiertfeld haben.

Rollen von Fenstern

Sieht man durch ein Fenster, so ist das Blickfeld normalerweise begrenzt. Handelt es sich bei dem Fenster um ein Microsoft-Windows-Fenster, so haben Sie Glück! Sie können das Fenster nach oben, unten, rechts oder links verschieben (rollen), um so mehr von der Grafik oder dem Text im Fenster zu sehen.

Meistens werden die Anwender die Bildlaufleisten an den Rändern des Client-Bereichs bedienen, um die aktuelle Anzeige zu rollen. Anders als die selbständigen Bildlaufleisten sind die Fensterbildlaufleisten Teil des Fensters selbst, angelegt für Fenster, die die Stilkonstanten *ws_VScroll* (vertikale Bildlaufleiste) oder *ws_HScroll* (horizontale Bildlaufleiste) bzw. beide Konstanten enthalten. Der Mechanismus hinter den Kulissen beruht auf einem Objekt mit dem Namen *Scroller*.

TScroller ist das ObjectWindows-Objekt, das die Fensterbildlaufleisten in Aktion setzt. Damit steht eine Möglichkeit zum Rollen der in das Fenster eingebrachten Grafiken und Texte zur Verfügung. Außerdem rollt *TScroller* das Fenster, wenn der Anwender die Maus aus dem Client-Bereich des Fensters hinausbewegt. Diesen Mechanismus bezeichnen wir als Automatisches Rollen. Er funktioniert auch für Fenster, die keine Bildlaufleisten haben.

Scroller-Attribute

Ein *TScroller* beinhaltet Werte, die festlegen, wieviel von einem Fenster gerollt werden soll. Diese Werte sind in den Objektfeldern von *TScroller* gespeichert, also in *XUnit*, *YUnit*, *XLine*, *YLine*, *XPage*, *YPage*, *XRange* und *YRange*. Felder, die mit *X* beginnen, repräsentieren horizontale Werte, Felder mit *Y* am Anfang stehen für vertikale Werte.

Eine Rolleinheit legt die kleinste Schrittweite des Rollvorgangs fest. Sie bestimmt die kleinste Anzahl von Geräteeinheiten (d.h. Pixel in einem Fenster, abhängig vom eingestellten Abbildungsmodus), in der vertikal oder horizontal gerollt werden kann. Die Einheit geht nomalerweise von der Art der Information aus, die angezeigt werden soll. Wenn Sie beispielsweise Text anzeigen, der eine Zeichenbreite von 8 und eine Zeichenhöhe von 15 Pixeln hat, sollten die Werte für *XUnit* und *YUnit* 8 bzw. 15 betragen.

Die weiteren Scroller-Attribute, *line* (Zeile), *page* (Seite) und *range* (Bereich) werden in Rolleinheiten ausgedrückt. *Line*- und *Page*-Werte geben die Anzahl der Einheiten an, um die als Antwort auf eine Anforderung des Anwenders gerollt werden soll. Wenn der Anwender einen der Pfeile am Ende der Bildlaufleiste anklickt, wird das Fenster entsprechend der im Feld *Line* gespeicherten Anzahl von Einheiten zeilenweise gerollt. Ein Klicken innerhalb der Bildlaufleiste selbst (nicht aber ein Anklicken des Bildlaufsymbols) hat ein seitenweises Rollen zur Folge. Die *Range*-Attribute repräsentieren die Gesamtanzahl der Einheiten, um die gerollt werden kann. Dieser Wert beruht normalerweise auf der Größe des Gegenstands im Fenster, wie zum Beispiel dem editierten Dokument.

Sehen wir uns nun als Beispiel ein Fenster zum Editieren von Text an. Wenn Sie eine Textdatei anzeigen wollen, die 400 Textzeilen mit einer Breite von 80 Zeichen pro Zeile und 50 Zeilen pro Seite hat, sollten Sie sinnvollerweise die in Tabelle 10.4 dargestellten Werte festlegen.

Ein *TScroller*-Objekt mit diesen Werten rollt um eine Zeile oder eine Seite weiter. Die gesamte Datei kann mit Hilfe der Bildlaufleisten oder des Automatischen Rollens durchgeblättert werden.

Tabelle 10.4 Typische Belegung für die Einstellungen eines Editierfensters

Feld	Wert	Bedeutung
XUnit	8	Zeichenbreite
YUnit	15	Zeichenhöhe
XLine, YLine	1	1 Einheit pro Zeile
XPage	40	40 Zeichen pro horizontaler Seite
YPage	50	50 Zeilen pro vertikaler Seite
XRange	80	Maximaler horizontaler Bereich
YRange	400	Maximaler vertikaler Bereich

Der voreingestellte Zeilenwert beträgt 1 und muß nur bei besonderen Wünschen geändert werden. Es gibt auch ein voreingestelltes Schema für das Setzen der Seiteneinheit, ausgehend von der aktuellen Größe des Fensters. In diesem Fall bedeutet das seitenweise Rollen ein Verschieben der Anzeige um die Höhe oder Breite des Client-Bereichs, abhängig von der Rollrichtung. Deshalb ist es nicht notwendig, daß Sie die Seitenwerte ändern, außer, wenn Sie diesen Mechanismus überladen wollen.

Dem Fenster einen Scroller zuweisen

Wenn Ihr Fenster über einen Scroller verfügen soll, konstruieren Sie ein *TScroller*-Objekt in Ihrem Fenster-Konstruktor und speichern das Objekt im Fensterobjektfeld *Scroller*. Dabei können die Größe der Einheiten und der Bereich festgelegt werden (falls Sie diese Werte wissen). Obwohl Fensterbildlaufleisten keinen Scroller benötigen (siehe das Automatische Rollen), verfügen viele rollbare Fenster trotzdem darüber. Wollen Sie ein Fenster mit Bildlaufleisten versehen, fügen Sie nur *ws_HScroll* oder *ws_VScroll* (oder beides) in das *Attr.Style*-Feld des Konstruktors ein. Im folgenden sehen Sie ein Beispiel für ein Fenster zum Editieren von Text:

```
constructor TScrollWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 400));
end;
```

T*Scroller.Init* hat als Argumente das rollbare Fenster und die Anfangswerte für die Felder *XUnit*, *YUnit*, *XRange* und *YRange*. Die Zeilen- und Seitenattribute sind auf deren Standardwerte gesetzt.

Bei der Anzeige des Fensters kann der Inhalt des Client-Bereichs vertikal oder horizontal mit Hilfe der Bildlaufleisten oder über das Automatische Rollen verschoben werden. Die *Paint*-Methode zeichnet die grafischen Informationen, ohne dabei berücksichtigen zu müssen, ob gerollt werden kann. E*s* ist selbst-

verständlich nicht effizient, ein großes Bild zu zeichnen, wenn nur ein Teil davon angezeigt werden kann. Die *Paint*-Methode kann jedoch dahingehend optimiert werden, daß nur die anzeigbaren Teile des Bildes gezeichnet werden. Mehr zu diesem Punkt erfahren Sie am Ende dieses Abschnitts.

Ein Roll-Beispiel

Wir werden Ihnen im folgenden Beispiel vorführen, wie eine vollständige Anwendung aufgebaut wird, die einen rollbaren Grafikentwurf zeichnet. Das Beispiel zeichnet eine Reihe von Rechtecken mit wachsender Größe, so daß das Bild nicht vollständig in den Client-Bereich eines Bildschirms mit einer gewöhnlichen VGA-Karte paßt. Mit Hilfe der Bildlaufleisten können verschiedene Teile des Bildes angezeigt werden. Sie können das Bild aber auch automatisch rollen, indem Sie den linken Mausknopf innerhalb des Client-Bereichs gedrückt halten und dann die Maus aus dem Client-Bereich bewegen.

Der Quelltext dazu sieht folgendermaßen aus (sie finden Ihn in der Datei SCROLAPP.PAS

```
program Scroll;

uses WObjects, WinTypes, WinProcs, Strings;

type
  TScrollApp = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

  PScrollWindow = ^TScrollWindow;
  TScrollWindow = object(TWindow);
    constructor Init(ATitle: PChar);
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
      virtual;
  end;

begin
  MainWindow := New(PScrollWindow, Init ('Boxes'));
end;

constructor TScrollWindow.Init(ATitle: PChar);
begin
  TWindow.Init(nil, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 60));
end;

procedure TScrollWindow.Paint(PaintDC: HDC; var PaintInfo:
  TPaintStruct);
var
  X1, Y1, I: Integer;
```

```
begin
  for I := 0 to 49 do
  begin
    X1 := 10 + I * 8;
    Y1 := 30 + I * 5;
    Rectangle(PaintDC, X1, Y1, X1 + X1, X1 + Y1 * 2);
  end;
end;

var
  ScrollApp: TScrollApp;

begin
  ScrollApp.Init('ScrollApp');
  ScrollApp.Run;
  ScrollApp.Done;
end.
```

Automatisches Rollen und Permanentes Aktualisieren

Das Automatische Rollen von *TScroller* ist standardmäßig aktiv. Sie können diese Eigenschaft ausschalten, wenn Sie das *AutoMode*-Objektfeld in *TScroller* auf *False* setzen. Das besitzende Fenster kann dies nach der Konstruktion des *TScroller*-Objekts in seinem Konstruktor durchführen:

```
Scroller := New(PScroller, Init(@Self, 8, 15, 80, 60));
Scroller^.AutoMode := False;
```

Nun kann nur noch mit Hilfe der Bildlaufleisten gerollt werden.

Ein weiterer Vorteil des Automatischen Rollens liegt darin, daß das Fenster immer schneller rollt, je weiter Sie die Maus aus dem Client-Bereich des Fensters bewegen. Abhängig davon, wie weit die Maus außerhalb liegt, rollt das Fenster entsprechend des kleinen Zeilen- oder des größeren Seitenwertes.

Neben dem Automatischen Rollen aktualisiert das obige Beispielprogramm permanent den Client-Bereich gemäß der Rollaktion, wenn der Bildlauf vom Anwender durch Verschieben des Bildlaufsymbols angefordert wird. Mit anderen Worten, das Bild wird entsprechend der Verschiebung des Bildlaufsymbols gerollt. Durch diese unmittelbare Rückmeldung gelangt der Anwender zum gewünschten Teil des Bildes, ohne den Mausknopf loslassen zu müssen. In einigen Situationen ist das Permanente Aktualisieren jedoch nicht wünschenswert.

Wenn Sie beispielsweise eine umfangreiche Textdatei anzeigen, würde das Permanente Aktualisieren die Geschwindigkeit herabsetzen, da immer wieder von Platte gelesen und der gewünschte Textabschnitt für jede Bewegung des Bildlaufsymbols angezeigt werden muß. In diesem Fall ist es günstiger, wenn Sie das Permanente Aktualisieren ausschalten:

```
Scroller^.TrackMode := False;
```

Während der Bewegung des Bildlaufsymbols findet kein Bildlauf statt. Erst wenn der Anwender die Maustaste losläßt, wird im Client-Bereich das Ergebnis der Verschiebung des Bildlaufsymbols dargestellt, d.h. der richtige Ausschnitt des Bildes wird angezeigt.

Ändern der Rolleinheiten und des Rollbereichs

In den bisherigen Beispielen sind wir davon ausgegangen, daß die Einheiten und die Bereichswerte zum Zeitpunkt der *TScroller*-Konstruktion bekannt sind. In vielen Fällen ist diese Information aber nicht vorhanden oder kann sich ändern, etwa wenn sich die Größe der anzuzeigenden Information ändert. Dann müssen die Bereichswerte (und eventuell auch die Einheiten) zu einem späteren Zeitpunkt gesetzt oder geändert werden. Kennen Sie bei der Konstruktion die Werte nicht, können Sie nil-Werte an den *TScroller*-Konstruktor übergeben.

Die *SetRange*-Methode übernimmt zwei Integerargumente: die Anzahl der horizontalen und der vertikalen Einheiten, die den gesamten Rollbereich festlegen. *SetRange* sollte immer dann verwendet werden, wenn sich die Größe des Bildes ändert. Nehmen wir an, Sie möchten ein Bild mit einer Breite von 100 Einheiten und einer Höhe von 300 Einheiten anzeigen. Folgender Befehl setzt den dafür korrekten Rollbereich:

```
Scroller^.SetRange(100, 300);
```

Sind die Einheiten bei der Initialisierung des *TScroller*-Objekts nicht bekannt, müssen ihre Werte gesetzt werden, bevor gerollt werden kann. Diese Werte könnten beispielsweise in der *SetupWindow*-Methode des Fensters angegeben werden:

```
procedure ScrollWindow.SetupWindow;
begin
  TWindow.SetupWindow;
  Scroller^.XUnit := 10;
  Scroller^.YUnit := 20;
end;
```

Ändern der Rollposition

Fenster können mit den Methoden *ScrollTo* und *ScrollBy* zum Rollen veranlaßt werden. Jede dieser Methoden hat zwei Integerargumente für die horizontalen und vertikalen Rolleinheiten. Wenn es zum Beispiel erforderlich wird, die Rollposition auf die obere linke Ecke des Bildes zu setzen, verwenden Sie *ScrollTo*:

```
Scroller^.ScrollTo(0, 0);
```

Wenn ein Bild in der vertikalen Richtung 400 Einheiten lang ist, kann die Rollposition folgendermaßen auf die Mitte des Bildes gesetzt werden:

```
Scroller^.ScrollTo(0, 200);
```

Die Methode *ScrollBy* verschiebt die Rollposition um eine bestimmte Anzahl von Einheiten nach oben, unten, links oder rechts. Negative Werte bewegen die Rollposition auf den Ursprungspunkt zu, also zur oberen linken Ecke, positive Werte verschieben die Position nach rechts unten. Wenn Sie den Client-Bereich um 10 Einheiten nach rechts und um 20 Einheiten nach unten rollen möchten, geben Sie folgenden Befehl ein:

```
Scroller^.ScrollBy(10, 20);
```

Bestimmen der Seitengröße

Per Voreinstellung wird die Seitengröße (*XPage* und *YPage*) entsprechend der Client-Bereichsgröße des Fensters festgelegt. Der Scroller wird benachrichtigt, wenn sich die Größe seines besitzenden Fensters ändert. Die Methode *WMSize* des besitzenden Fensters ruft die *SetPageSize*-Methode seines Scrollers auf, die dann die Objektfelder *XPage* und *YPage* besetzt. Die dabei verwendeten Werte basieren auf der Größe des Client-Bereichs und den Werten von *XUnit* und *YUnit*.

Wenn Sie diesen Mechanismus überschreiben und die Seitengröße direkt angeben wollen, müssen Sie die von Ihrem Fensterobjekt geerbte *WMSize*-Methode so überschreiben, daß *SetPageSize* nicht aufgerufen wird:

```
procedure TTestWindow.WMSize(var Msg: TMessage);
begin
  DefWndProc(Msg);
end;
```

Nun können Sie *XPage* und *YPage* direkt von Ihrem Fenster-Konstruktor aus festlegen (oder in einem von *TScroller* abgeleiteten Konstruktor):

```
constructor ScrollWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Style := Attr.Style or ws_VScroll or ws_HScroll;
  Scroller := New(PScroller, Init(@Self, 8, 15, 80, 400));
  Scroller^.XPage := 40;
  Scroller^.YPage := 100;
end;
```

Optimieren von Zeichenmethoden für das Rollen

Die obige Beispielanwendung zeichnet über 50 Rechtecke, versucht aber nicht festzustellen, ob alle Rechtecke auch im Client-Bereich des Fensters

angezeigt werden können. So kann es vorkommen, daß während des Programmlaufs viel Zeit und Mühe auf das Zeichnen von nicht anzeigbaren Grafiken verwendet wird. Die Funktion *TScroller.IsVisibleRect* kann in einer *Paint*-Methode eingesetzt werden, um das Zeichnen in Fenstern zu optimieren.

Die folgende *ScrollWindow.Paint*-Methode legt mit Hilfe von *IsVisibleRect* fest, ob die Windows-Funktion *Rectangle* aufgerufen wird oder nicht. *Rectangle* hat Argumente, die sich auf Geräteeinheiten beziehen, während *IsVisibleRect* sich auf Rolleinheiten bezieht. Aus diesem Grund müssen die Werte *X1* und *Y1* (der Ursprungspunkt des Rechtecks) und (*X2-X1*) und (*Y2-Y1*) (Breite und Höhe des Rechtecks) vor dem Aufruf von *IsVisibleRect* durch die jeweiligen Einheitswerte dividiert werden:

```
procedure TScrollWindow.Paint(PaintDC: HDC; var PaintInfo:
  TPaintStruct);
var
  X1, Y1, X2, Y2, I: Integer;
begin
  for I := 0 to 49 do
  begin
    X1 := 10 + I*8;
    Y1 := 30 + I*5;
    X2 := X1 + X1;
    Y2 := X1 + Y1 * 2;
    if Scroller^.IsVisibleRect(X1 div 8, Y1 div 15, (X2 - X1)
      div 8, (Y2 - Y1) div 15) then
      Rectangle(PaintDC, X1, Y1, X2, Y2);
    end;
end;
```

11. Dialogobjekte

Dialoge sind untergeordnete Fenster, die normalerweise für die Ausführung einer bestimmten Aufgabe angezeigt werden, wie beispielsweise zur Konfiguration eines Druckers oder zur Übernahme eines Texteintrags. Der Objekttyp *TDialog* unterstützt die Initialisierung, Erstellung und Ausführung sämtlicher Dialogfenstertypen. Wie bei *TWindow* können Sie für jeden Dialogtyp in Ihrer Anwendung Dialogtypen definieren, die von *TDialog* abstammen.

ObjectWindows unterstützt bereits die beiden häufigsten Dialoge, Texteingabe und Dateidialog. Außerdem wird der Typ *TDlgWindow* unterstützt, der einen Dialog erstellt, der sich eher wie ein normales Fenster verhält.

Verwendung von Dialogressourcen

Dialoge werden wie Menüs mit Hilfe einer Ressourcenbeschreibung entworfen und erstellt, die außerhalb des Programmcodes erzeugt wird. Die Dialogressource beschreibt das Aussehen und die Positionierung von Steuerelementen wie Schaltern, Listen, Editierbereichen und Textstrings. Sie beschreibt nur das Aussehen der Dialoge und nicht deren Verhaltensweisen. Dies liegt in der Verantwortung der Anwendung. Mit der Dialogressource ist ein Bezeichner verknüpft, der eine ID-Nummer (vom Typ Word) oder ein String (vom Typ PChar) sein kann. Innerhalb einer Anwendung wird das Dialogobjekt einer Dialogressource über die Ressourcen-ID zugeordnet.

Verwendung von Dialogobjekten

Die Verwendung von Dialogobjekten ähnelt der Verwendung von Pop-up-Fensterobjekten. Dialoge sind untergeordnete Fenster eines übergeordneten Fensters. Einfache Dialoge, die nur kurz angezeigt werden, können mit einer einzigen Methode des übergeordneten Fensterobjekts bearbeitet werden. Der Dialog kann mit einer Methode konstruiert, ausgeführt und freigegeben werden, wobei der Dialog nicht in einem Objektfeld gespeichert werden muß. Komple-

xere Dialoge werden Sie wahrscheinlich in einem Objektfeld Ihres Dialogtyps speichern.

So wie Pop-up-Fenster und Steuerelemente sind auch Dialoge untergeordnete Fenster. Sie werden bei der Konstruktion in die *ChildList* ihrer übergeordneten Fenster eingetragen.

Konstruieren und Initialisieren von Dialogen

Dialogobjekte werden mit einem *Init*-Konstruktor konstruiert und initialisiert. *Init* hat einen Zeiger auf das übergeordnete Fenster und einen PChar, der den Namen der Dialogressource repräsentiert, als Parameter:

```
ADlg := New (PSampleDialog, Init(@Self, 'EMPLOYEEINFO'));
```

oder

```
Dlg := New (PSampleDialog, Init(@Self, PChar(120));
```

Ausführen von Dialogen

Die Ausführung eines Dialogs läuft analog zur Erstellung und Anzeige eines Fensters. Da Dialoge normalerweise nur für einen relativ kurzen Zeitraum angezeigt werden, können einige Schritte abgekürzt werden. Dies hängt davon ab, ob der Dialog im modalen oder nicht-modalen Modus angezeigt werden soll.

Modale und nicht-modale Dialoge

Am häufigsten werden modale Dialoge verwendet. Ähnlich wie die mit der Windows-Funktion *MessageBox* erzeugten Meldungsfenster werden modale Dialoge zu einem ganz speziellen Zweck für einen kurzen Zeitraum angezeigt. *Modal* bedeutet, daß der Anwender während der Anzeige des Dialogs das übergeordnete Fenster weder anwählen noch verwenden kann. Der Anwender muß den Dialog verwenden und den *OK*- oder *Cancel*-Schalter anklicken, um den Dialog freizugeben und zu entfernen und mit dem Programm weiterzuarbeiten. Ein modaler Dialog friert die Operation des restlichen Programms ein. Daher können Sie mit einem Methodenaufruf, *ExecDialog*, einen Dialog erstellen, anzeigen und beenden:

```
ReturnValue := Application^.ExecDialog(ADlg);
if ReturnValue = id_OK then
  { Hier werden die Daten ermittelt und der Dialog bearbeitet }
else
  if ReturnValue = id_Cancel then  { Cancel wurde gedrückt }
```

Wenn der Anwender den *Cancel*-Schalter im Dialog betätigt, liefert *ExecDialog id_Cancel* zurück.

Ein nicht-modaler Dialog friert die Operation des Programms nicht ein. Er

kann – wie ein Fensterobjekt – in einem einzigen Schritt (mit *MakeWindow*) erstellt und angezeigt werden:

```
Application^.MakeWindow(ADlg);
```

Solange das Dialogobjekt existiert, können Sie jederzeit Daten aus einem Dialog ermitteln. Normalerweise wird dies in der *OK*-Methode durchgeführt, die dann aufgerufen wird, wenn der Anwender den *OK*-Schalter betätigt.

Beenden von Dialogen

Jeder Dialog muß dem Anwender eine Möglichkeit zu seiner Beendigung zur Verfügung stellen. Am häufigsten erfolgt dies mit dem *OK*- oder *Cancel*-Schalter (oder beiden). Nachkommen von *TDialog* antworten auf diese Schalter, indem sie die *EndDlg*-Methode aufrufen, die den Dialog beendet. Sie können aber auch andere Arten für die Beendigung eines Dialogs entwerfen. Einzige Bedingung ist, daß die *EndDlg*-Methode dabei aufgerufen wird. Sie können auch die *OK*- oder *Cancel*-Methode zur Änderung des Beendigungsverhaltens neu definieren.

So könnte beispielsweise eine *OK*-Methode definiert werden, die Anwendereingaben in einen Puffer außerhalb des Dialogobjekts kopiert. Wenn die Eingabe nicht gültig ist, können Sie ein Meldungsfenster anzeigen oder ein akustisches Signal erzeugen. Ist die Eingabe zulässig, rufen Sie *EndDlg* auf. Der an *EndDlg* übergebene Wert wird der Rückgabewert des Aufrufs von *ExecDialog*.

So wie Fensterobjekte rufen auch Dialogobjekte *CanClose* auf, bevor der Dialog geschlossen wird. Sie können *CanClose* überschreiben, um eine Beendigungsbedingung einzuführen, so wie bei dem Dialog, der die Anwendereingabe überprüft. Wenn Sie *CanClose* überschreiben, müssen Sie dort die geerbte *CanClose*-Methode aufrufen. Denn diese führt den Aufruf der *CanClose*-Methoden für die untergeordneten Fenster durch.

Verwalten von Dialogobjekten

Dialoge unterscheiden sich von anderen untergeordneten Fenstern (beispielsweise Pop-up-Fenstern und Steuerelementen) dadurch, daß Sie während der Lebensdauer ihrer übergeordneten Fenster häufig angezeigt und freigegeben sowie entfernt werden. Dialoge werden selten gleichzeitig mit ihren übergeordneten Fenstern angezeigt, freigegeben und entfernt. Normalerweise produziert ein Programm einen Dialog als Antwort auf eine Menüauswahl, auf ein Klicken mit der Maus, auf eine Fehlerbedingung oder ein anderes Ereignis.

Aus diesem Grund müssen Sie sicherstellen, daß nicht unzählige Dialoge erstellt werden, die dann nicht freigegeben werden. Denken Sie daran, daß alle konstruierten Dialogobjekte in die Listen der übergeordneten Fenster eingetragen werden, in denen die untergeordneten Fenster aufgeführt werden.

Bei modalen Dialogfenster ist es am günstigsten, wenn Sie alle Objekte in einer Methode des übergeordneten Fensters konstruieren, ausführen und frei-

geben, wie es in den Beispielen in diesem Kapitel gezeigt wird. Auf diese Weise ist jeder angezeigte Dialog ein neues Objekt.

Nicht-modale Dialoge ähneln eher Pop-up-Fenstern und Steuerelementen. Der ausschlaggebende Grund dafür, daß nicht-modale Dialoge (anders als modale Dialoge) nicht sofort nach ihrer Verwendung freigegeben werden dürfen, liegt darin, daß Sie nicht wissen können, wann der Anwender den Dialog schließt. (Bedenken Sie, daß bei modalen Dialogen die *ExecDialog*-Methode erst dann zurückkehrt, wenn der Dialog beendet wird.) Deshalb ist es am besten, wenn Sie nicht-modale Dialoge im Konstruktor des übergeordneten Fensters konstruieren und in einem Objektfeld desselben speichern. Anders als andere untergeordnete Fenster und Steuerelemente werden Dialoge nicht automatisch angezeigt, wenn die entsprechenden übergeordneten Fenster angezeigt werden.

```
constructor ParentWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
begin
  TWindow.Init(AParent, ATitle);
  ADlg := New(PSampleDialog, Init(@Self, 'EMPLOYEEINFO'));
end;
```

Soll der Dialog angezeigt werden, muß er jedesmal erstellt und angezeigt werden:

```
begin
  Application^.MakeWindow(ADlg);
end;
```

Schließlich wird jedes einzelne Dialogobjekt freigegeben, wenn sein übergeordnetes Fenster freigegeben wird.

Manipulieren von Steuerelementen und Bearbeiten von Botschaften

Auch die einfachsten Dialoge verfügen (als untergeordnete Fenster) über eine Reihe von Steuerelementen, wie Editierfelder, Listen und Schalter. Beachten Sie hierbei, daß diese Steuerelemente keine Steuerelementobjekte sind, sondern nur Schnittstellenelemente ohne Methoden und Objektfelder. In Kapitel 12 »Steuerelementobjekte« erfahren Sie, wie diese Objekte in einem Fenster (nicht in einem Dialog) eingesetzt werden. Im vorliegenden Kapitel stellen wir eine alternative Technik vor, mit deren Hilfe Sie Steuerelementobjekte mit den Steuerelementen eines Dialogs verknüpfen können (und zwar mit Hilfe von *InitResource*).

Es gibt eine zweigleisige Kommunikation zwischen einem Dialogobjekt und seinen Steuerelementen. In der einen Richtung muß der Dialog seine Steuerelemente manipulieren, zum Beispiel eine Liste ausfüllen. In der anderen Richtung muß er die Ereignisbotschaften seiner Steuerelemente bearbeiten und beantworten, wenn beispielsweise ein Anwender einen Eintrag in einer Liste auswählt.

Manipulieren von Dialog-Steuerelementen

Windows definiert Steuerelementbotschaften, die von der Anwendung an Windows zurückgesendet werden. So gehören zu den Listenbotschaften *lb_GetText*, *lb_GetCurSel* und *lb_AddString*.

Steuerelementbotschaften legen ein bestimmtes Steuerelement fest und übergeben Informationen in den Parametern *wParam* und *lParam*. Jedes Steuerelement in einer Dialogressource hat eine ID-Nummer, mit der Sie das Steuerelement bestimmen, das die Botschaft empfangen soll. Zum Senden einer Steuerelementbotschaft rufen Sie die *SendDlgItemMsg*-Methode von *TDialog* auf. Diese Methode füllt zum Beispiel eine Liste im Dialog mit einem Texteintrag aus, indem sie die Botschaft *lb_AddString* sendet:

```
procedure  TestDialog.FillListBox(var Msg: TMessage);
var
  TextItem: PChar;
begin
  TextItem := 'Item 1';
  SendDlgItemMsg(id_LB1, lb_AddString, 0, Longint(TextItem));
end;
```

Dabei ist *id_LB1* eine Konstante, die gleich der ID der Liste ist.

Wenn Sie ein Handle auf eines der Steuerelemente im Dialog benötigen, ermitteln Sie es mit der Methode *GetItemHandle*:

```
GetItemHandle(id_LB1);
```

Beantworten von Benachrichtigungsbotschaften der Steuerelemente

Wenn – in der anderen Kommunikationsrichtung – der Anwender ein Steuerelement anklickt, zum Beispiel einen Schalter betätigt oder aus einer Liste auswählt, empfängt der Dialog des Steuerelements eine spezielle, auf dem untergeordneten Fenster basierende Botschaft, eine sogenannte *Benachrichtigungsbotschaft* des Steuerelements. Definieren Sie im übergeordneten Dialogtyp jedes untergeordneten Steuerelements eine auf der untergeordneten ID basierende Botschafts-Antwortmethode:

```
TestDialog = object(TDialog)
  procedure HandleBN1Msg(var Msg: TMessage); virtual id_First +
  id_BN1;
  procedure HandleListBox(var Msg: TMessage); virtual id_First +
  id_LB1
end;
```

Jede Benachrichtigungsbotschaft des Steuerelements verfügt über einen Code, eine Integerkonstante, die die betreffende Aktion bezeichnet.

Die Auswahl eines Listeneintrags produziert zum Beispiel eine Botschaft mit dem Code *lbn_SelChange* (*lbn_* steht für »List Box Notification«). Das Anklicken eines Schalters produziert eine Botschaft mit dem Code *bn_Clicked* und die Eingabe in ein Editierfeld eine Botschaft mit dem Code *en_Changed*.

Es gibt Benachrichtigungscodes für Listen, Kombinationsfenster, Editierfelder und Schalter. Der Benachrichtigungscode wird in *Msg.lParamHi* an die

Botschaft übergeben. Zum Abfangen einer Benachrichtigungsbotschaft des Steuerelements schreiben Sie eine Antwortmethode für den Dialogtyp zur Bearbeitung der wichtigen Benachrichtigungscodes:

```
procedure TestDialog.HandleLB1Msg(var  Msg: TMessage);
begin
  case Msg.lParamHi of
    lbn_SelChange: {Bearbeitung der Auswahländerung}
    lbn_DblClk: {Bearbeitung des Doppelklicks};
  end;
end;
```

Beispiel für die Kommunikation zwischen Dialog- und Steuerelement

Im folgenden finden Sie ein Programm, dessen Hauptfenster ein modales Dialogfenster anzeigt, das mit dem Dialogtyp *TestDialog* definiert ist. Dieses Programm ist ein Beispiel für eine zweigleisige Kommunikation zwischen dem Dialogobjekt und seinen Steuerelementen. Die beiden *TestDialog*-Methoden, *HandleBN1Msg* und *HandleLB1Msg* sind auf der ID untergeordneter Fenster basierende Anwortmethoden, die aufgerufen werden, wenn der Anwender Steuerelemente (untergeordnete Fenster) anklickt. Wenn der Anwender zum Beispiel den *BN1*-Schalter ('Fill List Box') im Dialogfenster anklickt, wird die Methode *HandleBN1Msg* aufgerufen. Klickt der Anwender die Liste an, wird *HandleLB1Msg* aktiviert. In der »Gegenrichtung« sendet der Code der *HandleBN1Msg*-Methode dem Dialog eine Steuerelementbotschaft *lb_AddString*, um unter Verwendung der Dialogmethode *SendDlgItemMsg* die Liste mit Texteinträgen zu füllen.

Das Programm zeigt außerdem, wie ein neuer Dialog erzeugt wird, indem ein neuer Dialogtyp definiert und mit einer Dialogressource im Aufruf des Konstruktors *Init* in der *TextWindow.RunDialog*-Methode verknüpft wird. In Abbildung 11.1 sehen Sie das Dialogfenster.

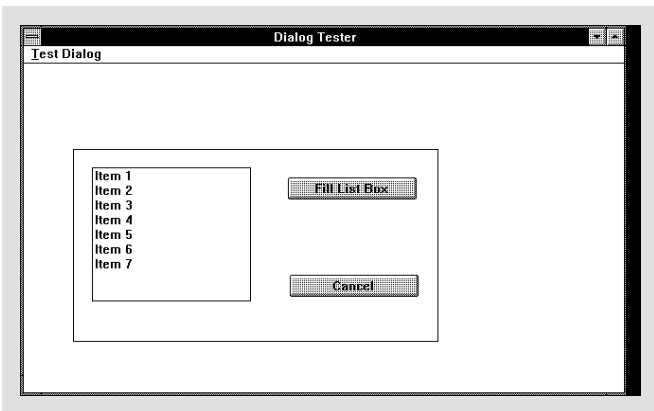


Abbildung 11.1
Eine Dialog-Anwendung

Das vollständige Programm finden Sie unter dem Namen DIALTEST.PAS auf Ihren Originaldisketten.

Zuordnen von Steuerelementobjekten

Bis jetzt haben Dialogobjekte auf Benachrichtigungsbotschaften der Steuerelemente mit Hilfe von Antwortmethoden geantwortet, die auf IDs untergeordneter Fenster basieren. Es gibt jedoch Fälle, in denen es günstiger ist, wenn das Steuerelement selbst auf eine Botschaft antwortet. So möchten Sie beispielsweise, daß in ein Editierfeld nur Zahlen eingegeben werden dürfen oder daß ein Schalter bei Betätigung den Stil wechselt. Dies ist bei Steuerelementobjekten in Fenstern ganz problemlos möglich (vgl. Kapitel 12 »Steuerelementobjekte«). Wenn Sie dieses Vorgehen auf Dialog-Steuerelemente anwenden wollen, die mit Ressourcendateien erzeugt wurden, müssen Sie einen anderen Konstruktor für die Konstruktion des Objekts verwenden.

Hierzu erstellen Sie ein Steuerelementobjekt, das Sie einem Steuerelement des Dialogs zuordnen. Dieses Objekt ermöglicht es Ihnen, Antworten auf Steuerelementbotschaften frei zu gestalten. Außerdem können Sie damit die Steuerelementobjekt-Methoden einsetzen, die in Kapitel 12 »Steuerelementobjekte« erläutert werden.

Im ersten Schritt muß ein Steuerelementobjekt definiert werden, das im Konstruktor des Dialogs konstruiert werden sollte. Verwenden Sie dazu jedoch – anstelle des *Init*-Konstruktors (wie es in Kapitel 12 beschrieben wird) – den Konstruktor *InitResource*, der die ID des übergeordneten Fensters und des Steuerelements (aus der Dialogressource) als Parameter benötigt. Dadurch werden die Botschafts-Antwortmethoden der Steuerelementobjekte aufgerufen und nicht die Standardbearbeitung für das Element. Danach können Sie Antwortmethoden für Ihr Steuerelementobjekt schreiben. Zu diesem Zweck müssen Sie einen neuen Objekttyp definieren, der von einem vorhandenen Steuerelementtyp abgeleitet wird.

Beachten Sie, daß Sie bei der Verwendung von *InitResource* jetzt auch das Steuerelement manipulieren können, indem Sie objektorientiert die Felder und Methoden des Objekts verwenden, anstatt Botschaften über die Windows-API an die Schnittstellenelemente senden zu müssen.

Beachten Sie außerdem noch, daß das Verknüpfen eines Objekts mit einem Steuerelement – anders als bei der Erstellung eines Fensterobjekts (das zwei Schritte, *Init* und *MakeWindow* umfaßt) – in einem Arbeitsschritt erledigt wird, da das Steuerelement bereits existiert: Es wird aus der Dialogressource geladen. Sie müssen *InitResource* nur mitteilen, *welches* Steuerelement – über seine ID – aus der Ressource mit dem Objekt verknüpft werden soll.

Aufruf von DefWndProc

Die letzte Anweisung in jeder Steuerelementobjekt-Methode sollte generell ein Aufruf von *DefWndProc(Msg)* sein, damit die standardmäßige Windows-Bearbeitung von Botschaften durchgeführt wird. Fügen Sie diesen Aufruf nicht ein, wird Ihre eigene Methode als *einzige Aktion* für die betreffende Botschaft ausgeführt. Dann muß Ihre Methode die Verantwortung für das Bearbeiten

(oder Ingnorieren) von Aufgaben übernehmen, die normalerweise standardmäßig ausgeführt werden. Dazu gehört beispielsweise die Anzeige von Schalterbetätigungen oder die Übergabe einer Botschaft an ein übergeordnetes Fenster des Steuerelements. Wenn Sie *DefWndProc* aufrufen, ergänzt Ihre Methode die normale Bearbeitung. Rufen Sie *DefWndProc* nicht auf, ersetzt Ihre Methode die standardmäßige Bearbeitung. Das Überschreiben der Standardbearbeitung ist für Sie dann von Vorteil, wenn eine Botschaft einen bestimmten Wert zurückliefert, Ihre Methode aber einen davon unterschiedlichen Wert zurückliefern soll.

Im folgenden sehen Sie eine Beispielanwendung, die über einen Dialog mit einem Schalter verfügt, der mit einem Objekt zum Ändern der Botschaftsantworten verknüpft ist. Der Schalter selbst erzeugt, wenn er gedrückt wird, ein akustisches Signal und ruft *DefWndProc* auf. Dadurch kann der Dialog die Botschaft ermitteln und anzeigen, wie oft der Schalter gedrückt wurde. Das Beispiel finden Sie in Abbildung 11.2.

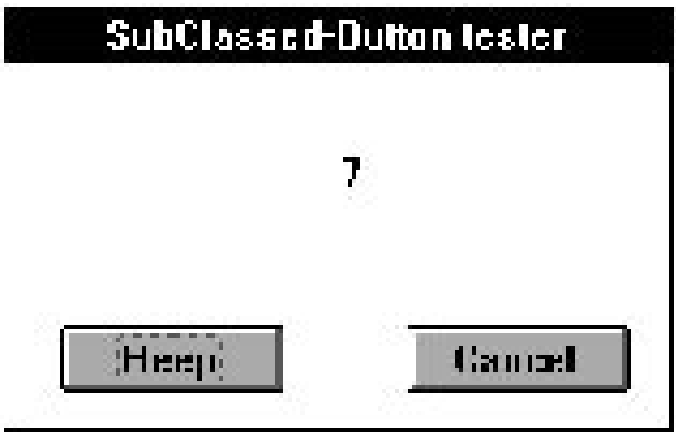


Abbildung 11.2 Ein Dialogfenster mit einem spezialisierten Schalter

Das vollständige Programm finden Sie unter dem Namen INIRTEST.PAS auf Ihren Originaldisketten.

Erweitertes Beispiel für einen Dialog

Im folgenden finden Sie ein erweitertes Beispiel, das die Definition eines komplexen Dialogs zeigt, das den Anwender zur Eingabe von Text auffordert und dann den Text auf Zulässigkeit überprüft. Das Programm zeigt einen Dialog an, der folgende Editierfelder enthält: Name des Beschäftigten, Sozialversicherungsnummer und ID des Beschäftigten. Wenn der Anwender *OK* anklickt, wird überprüft, ob der Name Ziffern enthält, ob die Versicherungsnummer die korrekte Länge hat und ob die ID eine Integerzahl ist.

Der neue Dialogtyp *EmployeeDlg* ändert die *CanClose*-Methode, um die Eingabe zu überprüfen.

In Abbildung 11.3 sehen Sie diese Anwendung.

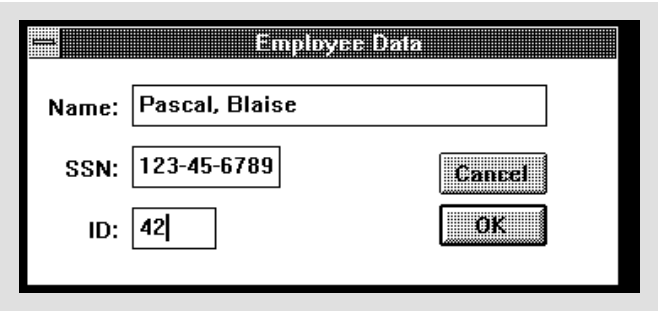


Abbildung 11.3 Ein Dialog, der die Anwendereingabe auf Zulässigkeit überprüft

Das vollständige Programm finden Sie unter dem Namen VDLGTEST.PAS auf Ihren Originaldisketten.

Dialogfenster

Nach der Arbeit mit Dialogen und Fenstern werden Sie sich vielleicht fragen, wo der Unterschied zwischen beiden liegt. Der Hauptunterschied besteht darin, daß einem Dialog eine Ressource zugeordnet ist, die den Typ und die Position der Steuerelemente festlegt. Aber auch ein Fenster kann über Steuerelemente verfügen. Eine Möglichkeit zur Plazierung von Steuerelementen in ein Fenster ist die Verwendung von Steuerelementobjekten, wie es in Kapitel 12 »Steuerelementobjekte« erläutert wird. Eine weitere Möglichkeit ist das Mischen von Fenster- und Dialogeigenschaften, wie es beim Objekttyp *TDlgWindow* der Fall ist. Dieser Typ erzeugt ein hybrides Objekt durch eine Vermischung der Eigenschaften von *TDialog* und *TWindow*. Diese zweite Möglichkeit ist der bequemere Weg zum Entwurf und zur Verwaltung vieler Steuerelemente in einem Fenster. Zusätzlich bietet diese Variante den Dialogen einige der flexibleren Fenstereigenschaften an.

TDlgWindow stammt von *TDialog* ab und erbt dessen Methoden, wie *Execute*, *Create, Ok* und *EndDlg*. Wie einfache Dialoge haben Dialogfenster eine korrespondierende Dialogressource. Andererseits haben diese Fenster, wie normale Fenster, eine verknüpfte Fensterklasse, die u.a. ein Symbol, einen Mauszeiger und ein Menü bestimmen kann. Aufgrund der Verknüpfung mit einer Fensterklasse sollte ein *TDlgWindow*-Nachkomme die Methoden *GetClassName* und *GetWindowClass* neu definieren, und der Klassenname sollte ebenfalls in der Dialogressource aufgeführt werden.

In den meisten Fällen werden Sie Dialogfenster wie normale Fenster oder nicht-modale Dialoge verwenden, d.h. mit den Methoden *Create* und *Show* anstelle der *Execute*-Methode.

Eine sinnvolle Verwendung für Dialogfenster ist ihr Einsatz als Hauptfenster einer Anwendung, wenn das Hauptfenster über viele komplexe Steuerelemente verfügen muß. Ein Kalkulationsprogramm kann beispielsweise ein Dialogfenster als Hauptfenster haben, in dem die Schalter für die Kalkulationsarten als Schalter-Steuerelemente in einer Dialogressource festgelegt sind. Dadurch wird

es möglich, daß das Hauptfenster über ein Menü, ein Symbol und einen Zeiger verfügen kann.

Eingabedialoge

Eingabedialoge sind vordefinierte Dialoge, die von ObjectWindows zur Verfügung gestellt werden. Dabei handelt es sich um einfache Dialogobjekte, die vom Typ *TInputDialog* definiert werden. Diese Objekte fordern den Anwender zur einer einzeiligen Texteingabe auf (vgl. Abbildung 11.4).

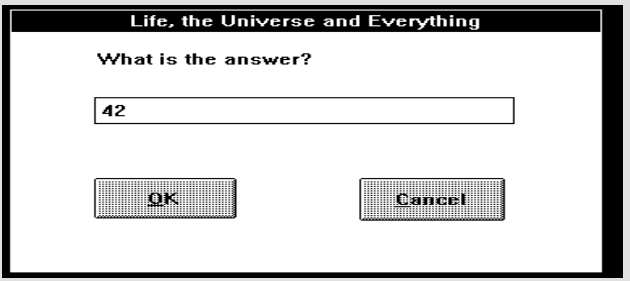


Abbildung 11.4 Ein Eingabedialog

Sie werden normalerweise Eingabedialoge als modale, seltener als nichtmodale Dialoge einsetzen. Verknüpft mit einem Eingabedialogobjekt ist eine Eingabedialogressource, die ObjectWindows in der Datei STDDLGS.RES zur Verfügung stellt. Stellen Sie sicher, daß Ihre Anwendung diese Ressource beinhaltet.

Bei der Verwendung der Unit *StdDlgs* werden die Ressourcen in STDDLGS.RES automatisch miteinbezogen.

Jedesmal, wenn Sie unter Verwendung der *Init-Methode* einen Eingabedialog konstruieren, legen Sie den Titel, die Eingabeaufforderung und einen Standardtext für den Dialog fest. Im folgenden sehen Sie den Aufruf des *Init*-Konstruktors eines Eingabedialogobjekts:

```
var SomeText: array[0..79] of Char;
begin
  AnInputDlg.Init(@Self, 'Caption', 'Prompt', SomeText,
    SizeOf(SomeText)))
  ...
end;
```

Im folgenden Beispiel ist *EditText* ein Textpuffer, der mit den Anwendereingaben gefüllt wird, sobald der Anwender den *OK*-Schalter anklickt.

Wenn der Anwender *OK* anklickt oder Return drückt, wird die in den Eingabedialog eingetragene Textzeile in das Zeichen-Array übertragen, das als Standardtext übergeben wurde. Das Beispiel konstruiert und zeigt einen Eingabedialog an und ermittelt den Text:

```
procedure TSampleWindow.Test(var Msg: TMessage);
var
  EditText: array[0..255] of Char;
begin
  EditText := 'Frank Borland';
  if ExecDialog(New(PInputDialog, Init(@Self, 'Data Entry',
    'Enter name:', EditText, SizeOf(EditText)))) = id_OK then
    MessageBox(HWindow, EditText, 'Name is:', mb_OK)
  else MessageBox(HWindow, EditText, 'Name is still:', mb_OK);
end;
```

Datei-Dialogfenster

Datei-Dialogfenster sind ein weiterer Typ von vordefinierten Dialogen, die vom ObjectWindows im Typ *TFileDialog* zur Verfügung gestellt werden. Verwenden Sie ein Datei-Dialogfenster immer dann, wenn Sie vom Anwender einen Dateinamen erfahren wollen, wie in den Anwendungsfunktionen *File Open*, *Save* und *Save As*. Sehen Sie sich dazu Abbildung 11.5 an.

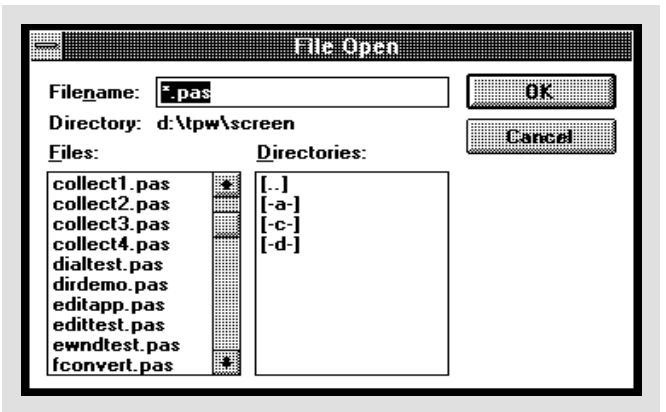


Abbildung 11.5
Ein Datei-Dialogfenster

In den meisten Fällen werden Datei-Dialogfenster als modale Fenster verwendet. Mit einem Datei-Dialogobjekt ist eine Datei-Dialogressource verknüpft, die von ObjectWindows in der Datei STDDLGS.RES zur Verfügung gestellt wird. Bei der Verwendung von *StdDlgs* wird die Ressourcendatei eingeschlossen.

Initialisieren von Datei-Dialogfenstern

TFileDialog.Init ermöglicht die Bestimmung eines übergeordneten Fensters (das festlegt, ob der Dialog ein Open- oder SaveAs-Dateidialog ist) und eines

nullterminierten Strings, der den Standardwert des editierten Textes an den Dialog übergibt. Derselbe String enthält den Dateinamen, wenn der Dialog beendet ist. Wird der Dialog abgebrochen, bleibt der String unverändert.

Im folgenden sehen Sie ein typisches Beispiel für einen Dialog zum Öffnen einer Textdatei:

```
var
  AFile: array[0..fsPathName] of Char;
begin
  StrCopy(AFile, '*.TXT');
  if Application^.ExecDialog(New(PFileDialog, Init(@Self,
    PChar(sd_FileOpen), AFile))) = id_OK then
    ...                        { AFile speichert den Dateinamen }
end;
```

Dateidialoge verwenden

Es gibt zwei Arten von Dateidialogen: den Dateidialog für das Öffnen und den für das Speichern einer Datei. Der Unterschied zwischen beiden beruht im Vorhandensein (bzw. Nichtvorhandensein) einer Liste mit Dateinamen.

Wenn Sie einen Dialog mit *ExecDialog* ausführen, erhalten Sie den Typ des Dialogs, der durch die Ressourcenvorlage (dem zweiten Parameter von *Init*) bestimmt wird. Enthält die Ressource eine Dateiliste (Steuerelement-ID *id_FList*), handelt es sich um einen Dialog zum Öffnen einer Datei. Ist die Steuerelement-ID *id_FList* nicht vorhanden, handelt es sich um einen Dialog zum Speichern von Dateien.

Sie können eigene Datei-Dialogressourcen erzeugen. Die Ressourcendatei STDDLGS.RES enthält zwei Vorlagen: eine Vorlage zum Öffnen (*sd_FileOpen*) und eine zum Speichern (*sd_FileSave*) einer Datei.

Im folgenden sehen Sie eine typische Verwendung eines Dateidialogs:

```
procedure TMyWindow.SaveAs(var AFile: array[0..fsPathName] of Char);
begin
  if Application^.ExecDialog(New(PFileDialog, Init(@Self,
    PChar(sd_FileSave), AFile))) = id_OK then
    begin
      { AFile enthält jetzt einen vollständigen Pfadnamen }
      SaveFile(AFile);              { speichert die Datei }
    end;
end;
```

12. Steuerelement-objekte

Die meisten Windows-Anwendungen machen Gebrauch von den Anwender-Schnittstellen, die unter dem Begriff Steuerelemente zusammengefaßt werden. In der folgenden Tabelle sind diejenigen Steuerelemente erläutert, die von den ObjectWindows-Objekttypen unterstützt werden, in Abbildung 12.1 sehen Sie die Bildschirmdarstellung typischer Steuerelemente.

Tabelle 12.1 Von ObjectWindows unterstützte Windows-Steuerelemente

Element	Objekttyp	Verwendung
Liste	*TListBox*	Eine rollbare Liste mit Einträgen, wie z.B. Dateien, aus denen ausgewählt werden kann.
Bildlaufleiste	*TScrollBar*	Eine selbständige Bildlaufleiste, ähnlich den Bildlaufleisten in Roll-Fenstern und Listen.
Aktionsschalter	*TButton*	Ein Aktionsschalter mit Text.
Auswahlfeld	*TCheckBox*	Ein Schalter mit Text, der ein- oder ausgeschaltet sein kann.
Schaltfeld	*TRadioButton*	Ein Schalter, der ein- oder ausgeschaltet sein kann; er wird normalerweise in Gruppen mit mehreren Schaltfeldern verwendet, die sich gegenseitig ausschließen.
Gruppe	*TGroupBox*	Ein statisches Rechteck mit Text in der oberen linken Ecke.
Editierfeld	*TEdit*	Ein Feld, in das Text eingegeben werden kann.
Statisches Steuerelement	*TStatic*	Anzeigbarer Text, den der Anwender nicht ändern kann.
Kombinationsfenster	*TComboBox*	Eine Kombination aus Liste und Editierfeld.
MDI-Client	*TMDIClient*	Vgl. Erläuterung in Kapitel 13 »MDI-Objekte«.

ObjectWindows definiert Objekttypen für alle in der vorhergehenden Tabelle aufgeführten Steuerelementtypen, die als untergeordnete Fenster innerhalb anderer Fenster verwendet werden. Steuerelemente, die zu Dialogfenstern gehören, sind in einer Dialogressource definiert und benötigen keine korrespondierenden ObjectWindows-Objekte (vgl. dazu Kapitel 11 »Dialogobjekte«). In diesem Kapitel wird die Funktionsweise der Steuerelementobjekte skizziert und erläutert, wie diese innerhalb eines Fensters verwendet werden können.

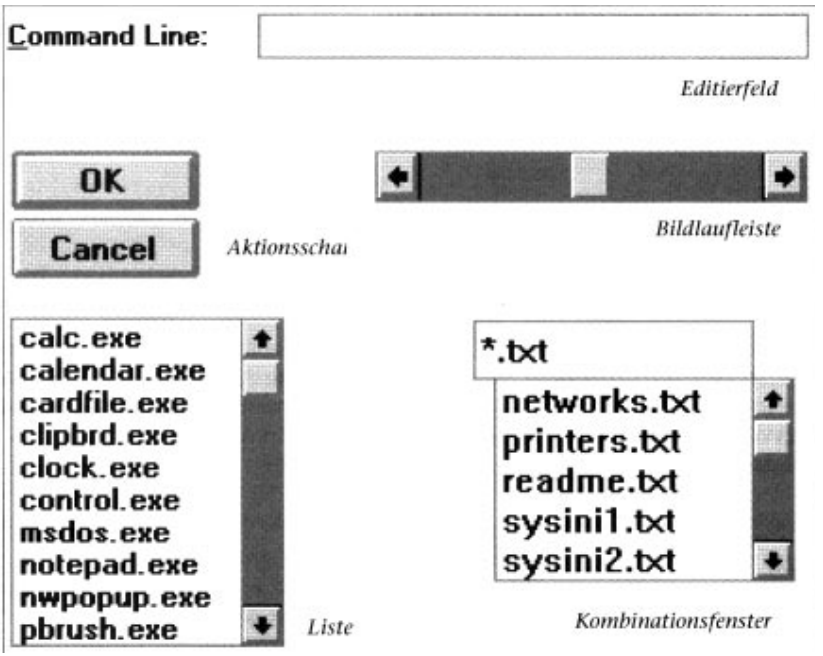


Abbildung 12.1 Beispiele für Steuerelemente

Der Objekttyp TControl

Für Windows sind Steuerelemente spezialisierte Fenster. In ObjectWindows stammt der Typ *TControl* von *TWindow* ab. Steuerelementobjekte ähneln den Fensterobjekten darin, daß sie ebenfalls erstellt, freigegeben und entfernt werden und als untergeordnete Fenster fungieren. Der Unterschied zu Fenstern liegt darin, wie sie auf Windows-Botschaften (*wm_*) antworten.

Sie werden sehen, daß die ObjectWindows-Steuerelementtypen alle Anforderungen erfüllen, die von Ihrer Anwendung gestellt werden. Daher können diese Typen sehr häufig unverändert angewendet werden. Sie können die vorhandenen Steuerelemente in einer unbeschränkten Anzahl von Kombinationen gemäß den von Ihrer Anwendung gestellten Anforderungen einsetzen. Es gibt aber auch Situationen, für die Sie Steuerelementtypen ableiten wollen. So können Sie z.B. einen speziellen Listentyp mit dem Namen *StatesListBox* definieren, der die Namen von 50 »Zuständen« speichert und bei Bedarf anzeigt.

Der Typ *TControl* ist genauso wie *TWindowsObject* ein abstrakter Objekttyp, dessen Nachkommen *TListBox*, *TButton*, etc. Sie instantiieren. Im folgenden Abschnitt werden das Verhalten und die Eigenschaften alle Steuerelementobjekte beschrieben.

Konstruieren und Erstellen von Steuerelementobjekten

Die Lebensdauer eines Steuerelements entspricht der Lebensdauer seines übergeordneten Fensters. In einem typischen Szenario definiert das übergeordnete Fenster ein Datenfeld für jedes untergeordnete Fenster. Wenn Sie einen Fenstertyp mit dem Namen *SampleWindow* erstellt haben, könnte dieser über ein Datenfeld mit dem Namen *LB1* verfügen, das ein *TListBox*-Objekt enthält.

Außer den Datenfeldern führen die übergeordneten Fenster im Feld *ChildList* eine Liste ihrer untergeordneten Fenster. Alle Steuerelemente sind untergeordnete Fenster und werden daher in diese Liste eingetragen, so daß hierfür keine Datenfelder erforderlich sind. Für den Zugriff und die Verwaltung von Steuerelementobjekten ist es gelegentlich günstiger, wenn diese entsprechende Objektfelder haben. Steuerelemente, die nur selten manipuliert werden, wie beispielsweise statische Steuerelemente oder Gruppen, sollten keine korrespondierenden Objektfelder erhalten. Bei änderungsanfälligeren Steuerelementen bleibt es Ihnen überlassen, ob Sie auf bestimmte Steuerelemente lieber mit einem Zeiger in seinem übergeordneten Fenster zugreifen wollen.

Zum Erstellen von Steuerelementen müssen zwei Arbeitsschritte ausgeführt werden. Der erste Schritt ist die Konstruktion des Steuerelementobjekts und der zweite die Erstellung des entsprechenden Steuerelements. Beide Schritte werden üblicherweise im Rahmen der Konstruktion und der Erstellung des übergeordneten Fensterobjekts ausgeführt. Die *Init*-Methode des übergeordneten Fensters ruft normalerweise den Konstruktor seiner untergeordneten Steuerelemente auf. So konstruiert zum Beispiel diese *Init*-Methode des Fenstertyps ein neues Listenobjekt und speichert es in einem Objektfeld mit dem Namen *LB1*:

```
constructor SampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
begin
  TWindow.Init(AParent, ATitle);
  LB1 := New(PlistBox, Init(@Self, id_LB1, 20, 20, 100, 80));
end;
```

Die sechs Felder im *Init*-Konstruktor der Liste stellen das übergeordnete Fensterobjekt des Steuerelements, seine ID, die X- und Y-Bildschirmkoordinaten seiner linken oberen Ecke, seine Breite (*W*) und seine Höhe (*H*) dar. Alle von ObjectWindows zur Verfügung gestellten Nachkommen von *TControl* benötigen zumindest diese sechs Argumente, mit Ausnahme von *TMDIClient* (vgl. dazu Kapitel 13 »MDI-Objekte«).

Steuerelementobjekte, zu denen es keine Objektfelder gibt, werden folgendermaßen konstruiert:

```
constructor TSampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var TempLBox: PListBox;
begin
  TWindow.Init(AParent, ATitle);
  TempLBox := New(PListBox, Init(@Self, id_LB1, 20, 20, 100,
```

```
      80));
end;
```

Alle Nachkommen von *TControl* erhalten – mit Ausnahme von *TMDIClient* – durch den Aufruf von *TControl.Init* die Stile *ws_Child* und *ws_Visible*. Wenn Sie den Stil eines Steuerelements ändern möchten, können Sie das Feld *Attr.Style* ändern, so wie es in Kapitel 11 »Dialogobjekte« für Fenster beschrieben ist.

Ein einem Steuerelementobjekt entsprechendes Steuerelement wird durch die vom übergeordneten Fenster geerbte *SetupWindow*-Methode erstellt. Sie müssen *Create* nicht explizit für die Steuerelementobjekte aufrufen.

Die Steuerelemente werden auch – falls erforderlich – in *SetupWindow* ausgefüllt und gesetzt. Im folgenden sehen Sie eine typische Implementierung der *SetupWindow*-Methode:

```
procedure SampleWindow.SetupWindow;
begin
  TWindow.SetupWindow;          { Erstellt die Steuerelemente }
  { Fügt Einträge in die Listen ein }
  LB1^.AddString('Item 1');
  LB1^.AddString('Item 2');
end;
```

Der Aufruf von *Show* ist für die Anzeige von Steuerelementen nicht erforderlich. Als untergeordnete Fenster werden sie mit ihren übergeordneten Fenstern angezeigt und neu gezeichnet. Sie können mit *Show* jedoch Steuerelemente ein- und ausblenden.

Um die Erstellung von Steuerelementen zusammenzufassen: Sie können optionalerweise Felder im Objekttyp Ihres übergeordneten Fensters definieren, die einige oder alle untergeordneten Steuerelemente beinhalten. Dann definieren Sie für Ihren Fenstertyp einen *Init*-Konstruktor, der das Steuerelementobjekt erstellt, und ein *SetupWindow*, das das Steuerelement aufbaut. Beachten Sie, daß alle Aufgaben von dem übergeordneten Fenster ausgeführt werden.

Freigeben und Entfernen von Steuerelementobjekten

Ebenso wie das Erstellen liegt auch das Freigeben von Steuerelementen im Aufgabenbereich des übergeordneten Fensters. Das Steuerelement wird gemeinsam mit dem übergeordneten Fenster entfernt, wenn der Anwender das Fenster oder die Anwendung schließt. Die Freigabe des entsprechenden Steuerelementobjekts wird vom Destruktor des übergeordneten Fensters durchgeführt.

Bearbeiten von Steuerelementen und Botschaften

Die Kommunikation zwischen einem Fensterobjekt und seinen Steuerelementobjekten ähnelt weitgehend der Kommunikation zwischen einem Dialo-

gobjekt und seinen Steuerelementen. Ebenso wie ein Dialog benötigt auch ein Fenster einen Mechanismus für die Manipulation seiner Steuerelemente und für die Beantwortung von Steuerelementereignissen, wie zum Beispiel der Auswahl aus einer Liste.

Manipulieren des Steuerelements eines Fensters

Dialoge manipulieren ihre Steuerelemente, indem sie mit der *SendDlgItemMsg*-Methode Botschaften an sie senden, und zwar mit Botschaftsnamen (wie beispielsweise *lb_AddString*) als Parameter (vgl. Kapitel 11 »Dialogobjekte«). Steuerelementobjekte vereinfachen diesen Prozeß jedoch durch Methoden wie *TListBox.AddString*, die die Steuerelemente direkt manipulieren.

Haben die Steuerelementobjekte eines Fensters entsprechende Objektfelder, ist es ganz einfach, die Steuerelement-Methoden aufzurufen:

```
LB1^.AddString('Scotts Valley');
```

Die Manipulation eines Steuerelementobjekts, das kein entsprechendes Objektfeld hat, macht es dagegen erforderlich, daß das Objekt aus der Liste des übergeordneten Fensters, in der die untergeordneten Fenster aufgeführt sind, abgerufen wird. Verwenden Sie dazu die *GetChildWithID*-Methode:

```
var
  TheListBox: PListBox;
begin
  TheListBox := PListBox(ChildWithID(id_LB1));
  TheListBox^.AddString('Scott Valley');
end;
```

Antwort auf Benachrichtigungsbotschaften der Steuerelemente

Ereignisse in den Steuerelementen eines Fensters erzeugen Benachrichtigungsbotschaften (vgl. dazu Kapitel 6 »Anzeigen von Fenstern«, Abschnitt »Antworten auf Steuerelement-Ereignisse«). In den meisten Fällen antworten Fensterobjekte auf Benachrichtigungsbotschaften von Steuerelementen durch den Aufruf von Antwortmethoden, die auf der ID des untergeordneten Fensters basieren.

Definieren Sie zu diesem Zweck im übergeordneten Fenstertyp für jedes untergeordnete Steuerelement Botschaftsantwort-Methoden, die auf der ID des untergeordneten Fensters basieren:

```
TSampleWindow = object(TWindow)
  LB1: PListBox;
  BN1, BN2: PButton;
  ST1: PStatic;
  procedure IDLB1(var Msg: TMessage); virtual id_First + id_LB1;
  procedure IDBN1(var Msg: TMessage); virtual id_First + id_BN1;
  procedure IDBN2(var Msg: TMessage); virtual id_First + id_BN2;
end;
```

In Ihren Antwortmethoden führt *Msg.lParamHi* den Steuerelement-Benachrichtigungscode, zum Beispiel *lbn_SelChange* oder *bn_Clicked*. Schreiben Sie eine Antwort, die die wichtigen Benachrichtigungscodes verwaltet:

```
procedure TSampleWindow.IDLB1(var Msg: TMessage);
begin
  case Msg.lParamHi of
    lbn_SelChange:  { Bearbeitung der Auswahländerung };
    lbn_DblClk:     { Bearbeitung des Doppelklicks };
  end;
end;
```

Es kann Situationen geben, in denen ein Steuerelementobjekt selbst auf eine Benachrichtigungsbotschaft antworten soll. Bauen Sie hierzu ein derartiges Verhalten in das Steuerelement ein. So könnten Sie beispielsweise einen Schalter entwerfen, der seinen Text ändert, wenn er vom Anwender angeklickt wird, oder ein Editierfeld anlegen, das nur die Eingabe von Ziffern zuläßt.

Derartige Eigenschaften können Sie für Steuerelemente in Fenstern definieren, weil diese Steuerelementobjekte sind. Steuerelemente in Dialogfenstern können nur dann auf Benachrichtigungsbotschaften antworten, wenn sie mit Steuerelementobjekten verknüpft sind (über *InitResource*).

In diesen speziellen Fällen wird es Ihnen lieber sein, wenn der Steuerelementtyp das Verhalten übernimmt, anstatt es in jedes übergeordnete Fenster zu kopieren. Das Ergebnis ist ein selbständiger Objekttyp, der in vielen Anwendungen wieder verwendet werden kann.

Um ein Steuerelementobjekt so zu programmieren, daß es seine Benachrichtigungsbotschaften direkt beantwortet, definieren Sie für seinen Typ eine benachrichtigungs-basierende Anwortmethode (anstelle einer auf der ID des untergeordneten Fensters basierenden Antwort für sein übergeordnetes Fensterobjekt). Verwenden Sie die Summe von *nf_First* und dem Steuerelement-Benachrichtigungscode als Bezeichner im Methodenkopf. Definieren Sie zu diesem Zweck einen Typ, der von einem vorhandenen Steuerelementtyp abstammt:

```
TSpecializedLB = object(TListBox)
  procedure LBNSelChange(var Msg: TMessage); virtual nf_First +
    lbn_SelChange;
  procedure LBNDblClk(var Msg: TMessage); virtual nf_First +
    lbn_DblClk;
end;

procedure TSpecializedLB.LBNSelChange(var Msg: TMessage);
begin
  { Bearbeitung der Auswahländerung };
  Msg.Result := 1;
end;
```

Wenn Sie wollen, daß auch das übergeordnete Fenster eine Benachrichtigungsbotschaft empfängt, nachdem das Steuerelement sie bearbeitet hat, muß dieses die Botschaft explizit an sein übergeordnetes Fenster weitergeben. Hierzu muß lediglich die Routine *DefNotificationProc* aufgerufen werden.

Fenster, die wie Dialoge agieren

Ein Dialogfenster mit Steuerelementen läßt den Anwender mit der Tab-Taste durch alle Steuerelemente »wandern«. Außerdem können in diesen Fenstern mit den Pfeiltasten die Schaltfelder in einer Gruppe gewählt werden. Um diese Tastaturschnittstelle für Fenster mit Steuerelementen zu emulieren, rufen Sie die *TWindowsObject*-Methode *EnableKBHandler* für das Fensterobjekt in seinem Konstruktor auf.

Listen

Eine Liste ist die einfachste Art, den Anwender eines Windows-Programms zur Auswahl eines Listeneintrags aufzufordern. Abhängig von der Art des Programms muß der Anwender in Listen mit Dateien, Druckern, Formen oder Früchten eine Auswahl treffen. Listen sind im Objekttyp *TListBox* gekapselt. *TListBox* definiert vier Methoden zur Ausführung folgender Aufgaben: Erstellen von Listen, Ändern der Listeneinträge, Abfragen der Listeneinträge und Feststellen, welcher Eintrag vom Anwender ausgewählt wurde.

Konstruieren und Erstellen von Listen

Der *Init*-Konstruktor von *TListBox* hat ein übergeordnetes Fenster, eine ID und die Steuerelementmaße *X*, *Y*, *W* und *H* als Argumente:

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
```

TListBox.Init ruft *TControl.Init* auf, das dem Steuerelement den Stil *ws_Child* oder *ws_Visible* zuweist. Danach wird der Stil *lbs_Standard* hinzugefügt. *lbs_Standard* ist eine Kombination aus *lbs_Notify* (zum Empfang von Benachrichtigungsbotschaften), *ws_VScroll* (für eine vertikale Bildlaufleiste), *lbs_Sort* (zur alphabetischen Sortierung der Listeneinträge) und *ws_Border* (für einen Rahmen). Wenn Sie einen anderen Listenstil verwenden wollen, modifizieren Sie das *Attr.Style*-Feld von *TListBox*. Sollen beispielsweise die Einträge in einer Liste nicht alphabetisch sortiert werden, schreiben Sie folgenden Code:

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
LB1^.Attr.Style := LB1^.Attr.Style and not lbs_Sort;
```

oder

```
LB1 := New(PListBox, Init(@Self, id_LB1, 20, 20, 340, 100));
LB1^.Attr.Style := ws_Child or ws_Visible or lbs_Notify or
  ws_VScroll or ws_Border;
```

Wie alle untergeordneten Fenster werden auch Listenobjekte von ihren übergeordneten Fenstern erstellt.

Ändern von Listen

Nach der Erstellung einer Liste, müssen Sie diese mit Einträgen (mit Strings) auffüllen. Im weiteren Verlauf der Arbeit werden Sie Einträge hinzufügen oder entfernen oder die gesamte Liste löschen wollen. Zum Eintragen und Hinzufügen von Einträgen rufen Sie die Listenmethode *AddString* auf:

```
LB1^.AddString('Item 1');
LB1^.AddString('Item 2');
LB1^.AddString('Item 3');
LB1^.AddString('Item 4');
LB1^.AddString('Item 5');
LB1^.AddString('Item 6');
```

Ist *AddString* nicht erfolgreich, liefert es einen negativen Wert zurück.

Eine Liste speichert seine Strings ähnlich wie ein Array aus Strings mit Hilfe eines Index. Wenn *LB1* ein Array wäre, würde 'Item 1' an die Position *LB1[0]* gespeichert, 'Item 2' an die Position *LB1[1]*, 'Item 3' an die Position *LB1[2]*, etc. Bei jedem Aufruf von *AddString* wird der angegebene String per Voreinstellung in alphabetischer Reihenfolge eingefügt oder an das Ende der Liste angefügt, wenn der Stil *lbs_Sort* nicht beinhaltet.

Unabhängig davon, ob die Liste sortiert ist, können Sie ein neues Element mit *InsertString* an eine bestimmte Indexposition einfügen:

```
LB1^.InsertString('Item 1.5', 1);
```

Dieser Befehl verschiebt die Strings an den Indexpositionen, die größer oder gleich 1 sind, um eine Stelle. Die obigen sieben Methodenaufrufe ergeben folgende Liste (sehen Sie sich dazu Abbildung 12.2 an):

Index	Eintrag
0	'Item 1'
1	'Item 1.5'
2	'Item 2'
3	'Item 3'
4	'Item 4'
5	'Item 5'
6	'Item 6'

Das Einfügen eines neuen Strings an der Indexposition -1 fügt den String an das Ende der Liste an.

Mit *DeleteString* können Sie einen Eintrag entfernen. Der folgende Methodenaufruf löscht den String an der Position 1 ('Item 1.5') und verschiebt die Strings an den Positionen über 1 um eine Stelle nach unten:

```
LB1^.DeleteString(1);
```

ClearList löscht alle Strings in der Liste:

```
LB1^.ClearList;
```

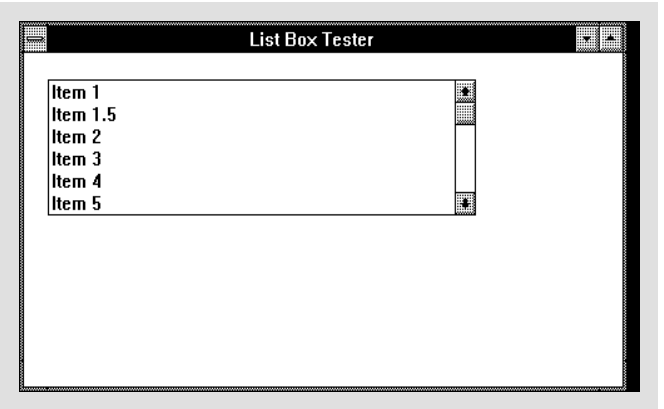


Abbildung 12.2
Eine ausgefüllte Liste

Abfragen von Listen

Mit Hilfe von drei Methoden können Sie sich Informationen über die in einem Listenobjekt enthaltene Liste verschaffen.

GetCount liefert die Anzahl der Einträge in der Liste zurück.

GetString ermittelt den String in der Liste, der an der durch das Integerargument angegebenen Position im Index steht. Das PChar-Argument zeigt auf einen Puffer, der den String aufnehmen soll. *GetString* liefert auch einen Integerwert zurück, der die Länge des Strings angibt.

GetStringLen liefert die Länge des Strings an der angegebenen Position zurück, den String selbst jedoch nicht.

Aus einer Liste auswählen

Es gibt nur wenig, was der Anwender mit einer Liste tun kann: Er kann durch die Liste rollen und einen Eintrag einmal oder zweimal anklicken. Wenn in einer Liste eine Anwenderaktion ausgeführt wird, sendet Windows eine *Listen-Benachrichtigungsbotschaft* an das übergeordnete Fenster der Liste. Normalerweise definieren Sie eine auf der ID des untergeordneten Fensters basierende Methode im übergeordneten Fenstertyp, die die Botschaften für jedes Steuerelement des übergeordneten Fensters bearbeitet.

Jede Listen-Benachrichtigungsbotschaft (*lbn*) enthält einen Listen-Benachrichtigungscode (eine Integerkonstante) in *Msg.lParamHi*, der die Art der Aktion angibt.

In der folgenden Tabelle finden Sie eine Beschreibung der gebräuchlichsten *lbn*-Codes:

Tabelle 12.2 Listen-Benachrichtigungsbotschaften

wParam	Aktion
lbn_SelChange	Es wurde eine Option mit einem einzigen Maus-Klick gewählt.
lbn_DblClk	Es wurde eine Option mit einem doppelten Maus-Klick gewählt.
lbn_SetFocus	Der Anwender hat der Liste durch einmaliges oder doppeltes Anklicken eines Eintrags oder durch die Verwendung der Tab-Taste den Fokus übergeben. *lbn_SetFocus* kommt vor der *lbn_SelChange*-Botschaft.

Im folgenden sehen Sie die Methode eines übergeordneten Fensters, die die Listenbotschaften verwaltet:

```
procedure TLBoxWindow.HandleLB1Msg(var Msg: TMessage);
var
  Idx: Integer;
  ItemText: string[10];
begin
  if Msg.lParamHi = lbn_SelChange then
  begin
    Idx := LB1^.GetSelIndex;
    if LB1^.GetStringLen(Idx) < 11 then
    begin
      LB1^.GetSelString(@ItemText, 10);
      MessageBox(HWindow, @ItemText, 'You selected:', mb_OK);
    end;
  end;
  else DefWndProc(Msg);
end;
```

Der Anwender hat eine Auswahl getroffen, wenn *Msg.lParamHi* der *lbn*-Konstante *lbn_SelChange* entspricht. Ist dies der Fall, wird die Länge des ausgewählten Strings festgestellt, überprüft, ob der String in den 10 Zeichen langen String paßt, der String ermittelt und in einem Meldungsfenster angezeigt (vgl. Abbildung 12.3).

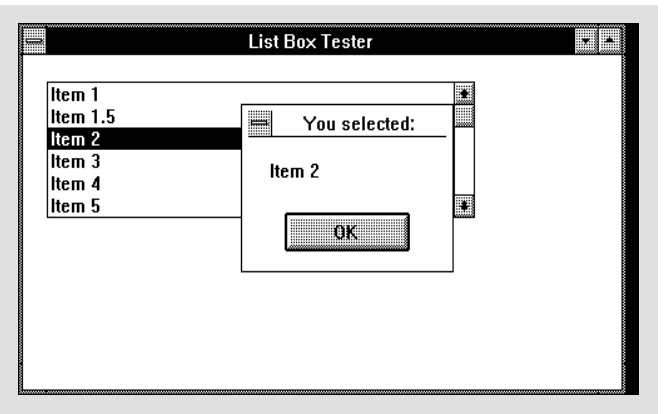


Abbildung 12.3 Antwort auf die Anwender-Auswahl eines Listeneintrags

Ausgewählte Listeneinträge werden von vier *TListBox*-Methoden manipuliert: *GetSelString*, *GetSelIndex*, *SetSelString* und *SetSelIndex*. Die *Get*-Methoden ermitteln den String und die Indexposition des ausgewählten Strings. Die *Set*-Methoden übergehen den Anwender und bewirken durch Bestimmung des Strings oder der Indexposition die Auswahl eines bestimmten Eintrags. Dadurch wird der String auch in den sichtbaren Bereich gebracht, wenn dies nicht bereits der Fall ist.

Beispielprogramm: LBoxTest

Das Programm *LBoxTest* ist ein vollständiges Programm, das ein Fenster mit einer Liste erstellt. Beim Start der Anwendung wird das Hauptfenster mit einer Liste angezeigt. Wenn der Anwender einen Listeneintrag auswählt, wird ein Dialogfenster mit dem ausgewählten Eintrag eingeblendet. Beachten Sie die Beziehung zwischen dem Fenster- und dem Listenobjekt. Die Liste ist nicht nur ein untergeordnetes Fenster des Hauptfensters, sondern sie *gehört* zum Hauptfenster auch als Objektfeld. Eines der Objektfelder des Hauptfensters ist *LB1*, das das Listenobjekt enthält. Die vollständige Datei finden Sie unter dem Namen LBOXTEST.PAS auf Ihren Originaldisketten.

Statische Steuerelemente

Statische Steuerelemente sind normalerweise unveränderbare Texteinheiten oder einfache Grafiken, die in einem Fenster oder Dialog auf dem Bildschirm angezeigt werden. Der Anwender kommuniziert nicht mit diesen Elementen, obwohl das Programm deren Text ändern kann. In Abbildung 12.4 sehen Sie verschiedene statische Elemente und ihre entsprechenden Windows-Stilkonstanten.

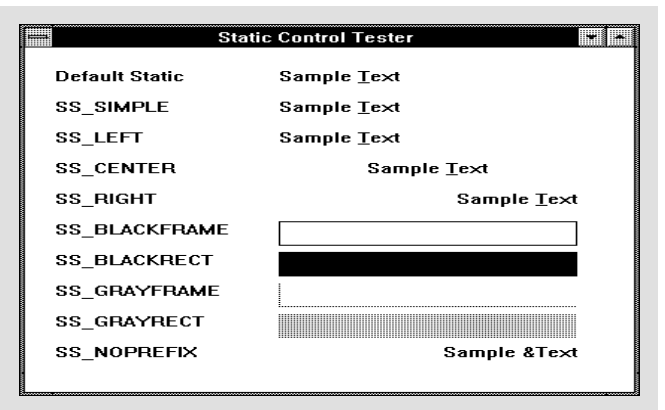


Abbildung 12.4 Statische Elemente

Konstruieren statischer Steuerelemente

Da der Anwender niemals direkt mit statischen Steuerelementen umgeht, empfängt die Anwendung selten (wenn überhaupt) Benachrichtigungsbotschaften, die ein statisches Steuerelement betreffen. Aus diesem Grund können diese Elemente mit -1 oder mit einer anderen ungebräuchlichen Zahl als ID-Ressource konstruiert werden – wobei Sie dafür immer dieselbe Zahl verwenden können.

Zusätzlich zu den üblichen *Init*-Parametern für Steuerelementobjekte hat *TStatic.Init* einen weiteren Parameter für die Textlänge, der die maximale Länge des Textes festlegt, die in dem Steuerelement Platz findet.

Da der Text eine terminierende 0 enthalten muß, ist die Anzahl der tatsächlich anzeigbaren Zeichen um 1 niedriger als die an den Konstruktor übergebene Anzahl.

Der *Init*-Konstruktor von *TStatic* konstruiert neue statische Steuerelementobjekte.

```
Stat1 := New(PStatic, Init(@Self, id_ST1, '&Text', 20, 50, 200,
  24, 6));
```

In den meisten Fällen werden Sie auf ein statisches Steuerelementobjekt nach seiner Erstellung weder zugreifen noch es ändern. Aus diesem Grund ist die Zuweisung eines Objektfeldes, das das statische Steuerelementobjekt enthält, unnötig. Eine temporäre Variable reicht für diesen Zweck aus:

```
TempStat := New(PStatic, Init(@Self, id_ST1, '&Text', 20, 50, 200,
  24, 6));
```

Init produziert ein statisches Steuerelement mit dem Stil *ss_Left* (links ausgerichtet) sowie mit *ws_Child* und *ws_Visible*. Zur Änderung des Stils modifizieren Sie das Feld *Attr.Style*:

```
Stat1^.Attr.Style := Stat1^.Attr.Style and (not ss_Left) or
  ss_Center;
```

Eine für statische Elemente verfügbare Option ist das Unterstreichen eines oder mehrerer Zeichen im Textstring. Die Implementierung und ihr Ergebnis ähneln der Unterstreichung des ersten Zeichens einer Menüauswahl: Fügen Sie ein &-Zeichen unmittelbar vor das Zeichen ein, das unterstrichen werden soll. Wenn Sie zum Beispiel das T in 'Text' unterstreichen wollen, senden Sie den String '&Text' im *Init*-Aufruf.

Soll das &-Zeichen als solches in einem String verwendet werden, müssen Sie den statischen Windows-Stil *ss_NoPrefix* einsetzen (vgl. Abbildung 12.4).

Informationen über statische Steuerelemente

Mit der *GetText*-Methode können Sie sich über den Text informieren, der in einem statischen Steuerelement steht.

Ändern von statischen Steuerelementen

TStatic verfügt über zwei Methoden, mit deren Hilfe Sie den Text eines statischen Steuerelements ändern können. *SetText* setzt den Text auf denjenigen, der von dem *Char*-Argument übergeben wird. *Clear* löscht den Text eines statischen Steuerelements.

Text in Elementen, die mit dem Stil *ss_Simple* erzeugt wurden, kann jedoch nicht geändert werden.

Beispiel: Die Anwendung StatTest

Das Programm *StatTest* erstellt die Test-Anwendung für statische Elemente, die Sie in Abbildung 12.4 sehen. Beachten Sie, daß die Beschriftungen wie *Default Static* und *ss_Simple* ebenso statische Steuerelemente sind wie *Sample Text* und die schwarzen und grauen Felder.

Die vollständige Datei finden Sie unter dem Namen STATTEST.PAS auf Ihren Originaldisketten.

Aktionsschalter-Steuerelemente

Die Aktionsschalter (manchmal auch »Befehlsschalter« genannt) führen bei jeder Betätigung bestimmte Aufgaben durch. Es gibt zwei Stile für Aktionsschalter, die beide vom Typ *TButton* sind: *bs_PushButton* und *bs_DefPushButton*. Voreingestellte Aktionsschalter haben einen dicken Rahmen und werden für voreingestellte Anwenderantworten verwendet. In Abbildung 12.5 sehen Sie ein Windows-Beispielprogramm, das reguläre und voreingestellte Aktionsschalter einsetzt.

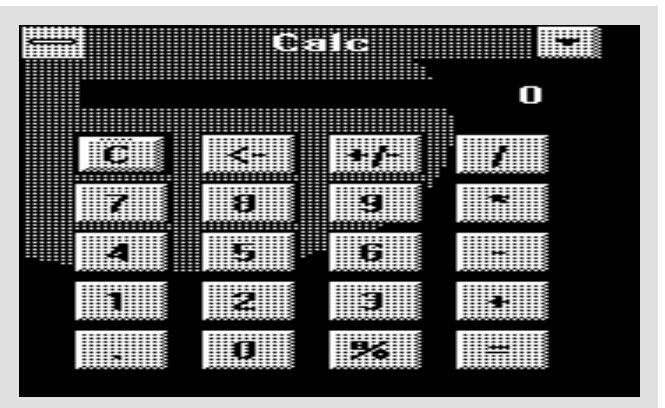


Abbildung 12.5 Ein Windows-Programm mit Aktionsschaltern

Der *Init*-Konstruktor von *TButton* benötigt das übergeordnete Fenster, die Steuerelement-ID, einen Textstring vom Typ PChar, die Position und Größe der Schalter und das Boolesche Flag *IsDefaultButton* als Argumente. *IsDefaultButton* zeigt an, ob der Schalter ein voreingestellter oder regulärer Aktionsschalter ist. Ein voreingestellter Aktionsschalter hat eine schwarze Umrahmung und wird standardmäßig beim Drücken von Return aktiviert.

```
Push1 := New(PButton, Init(@Self, id_Push1, 'Test Button', 38,
  48, 316, 24, False));
```

Beantworten von Schalterbotschaften

Sobald der Anwender einen Aktionsschalter drückt, empfängt das übergeordnete Fenster eine auf der ID des untergeordneten Fensters basierende Botschaft. Wenn das übergeordnete Fensterobjekt die Botschaft abfängt, gibt es verschiedene Antwortmöglichkeiten, wie Öffnen eines Dialogfensters, Speichern einer Datei, etc. Zur Antwort auf Schalterbotschaften definieren Sie eine auf der ID des untergeordneten Fensters basierende Methode, die alle betreffenden Schalter bearbeitet. Windows stellt nur einen Benachrichtigungscode für Aktionsschalter, *bn_Clicked*, zur Verfügung, sie müssen ihn daher nicht überprüfen.

```
TTestWindow = object(TWindow)
  But1: PButton;
  procedure IDBut1(var Msg: TMessage); virtual id_First +
    id_But1;
  ...
end;

procedure TestWindow.IDBut1(var Msg: TMessage);
begin
  MessageBox(HWindow, 'Clicked', 'The Button was:', mb_OK);
end;
```

Auswahl- und Schaltfelder

Der Typ *TCheckBox* stammt von *TButton* und der Typ *TRadioButton* von *TCheckBox* ab. Auswahlfelder lassen den Anwender zwischen zwei Möglichkeiten wählen, d.h. der Anwender kann das Steuerelement als ein- oder ausgeschaltet markieren, bzw. es so belassen, wie es eingestellt ist. Sind mehrere Auswahlfelder zu einer Gruppe zusammengefaßt, können mehrere Felder als eingeschaltet markiert sein. So können Auswahlfelder beispielsweise für die Bestimmung von drei Schriften eingesetzt werden, die eine Anwendung laden soll. Schaltfelder werden dagegen für die Auswahl einer von mehreren Optionen

verwendet, die sich gegenseitig ausschließen. So werden Schaltfelder für die Bestimmung einer Schrift für ein einzelnes Zeichen verwendet. Wir fassen Auswahl- und Schaltfelder manchmal unter dem Oberbegriff *Wahlfelder* zusammen.

Ein wichtiger Aspekt von Wahlfeldern ist deren Status. Während der Anzeige auf dem Bildschirm ist ein Wahlfeld entweder ein- oder ausgeschaltet. Das Anklicken eines Wahlfeldes durch den Anwender stellt ein Ereignis dar, das eine Windows-Botschaft erzeugt. Wie bei anderen Steuerelementen werden diese Botschaften normalerweise vom übergeordneten Fenster des Wahlfeldes abgefangen und bearbeitet. Sie können von *TCheckbox* und *TRadioButton* auch Typen ableiten, die bei Betätigung eine bestimmte Aktion ausführen. Wenn Ihr Typ eine Methode für *nf_First + bn_Clicked* definiert, sollte diese als erstes die Antwortmethode *BNClicked* seines Vorfahren aufrufen und dann die zusätzlich gewünschten Aktionen ausführen.

Konstruieren von Auswahl- und Schaltfeldern

Die *Init*-Konstruktoren von Auswahl- und Schaltfeldern benötigen eine Steuerelement-ID, einen Textstring, Positionierung, Größe und *AGroup* als Argumente. *AGroup* ist ein Zeiger auf ein Gruppenobjekt (vgl. dazu den Abschnitt »Gruppen« weiter hinten in diesem Kapitel), das Auswahl- und Schaltfenster logisch gruppiert. Wenn *AGroup* den Wert **nil** hat, ist das Wahlfeld kein Teil einer logischen Gruppe.

```
GroupBox1 := New(PGroupBox, Init(@Self, id_GB1, 'A Group Box',
  38, 102, 176, 108));
ChBox1 := New(PCheckBox, Init(@Self, id_Check1,
  'Check Box Text', 235, 12, 150, 26, GroupBox1));
```

Auswahlfelder werden standardmäßig mit dem Stil *bs_AutoCheckBox*, Schaltfelder mit dem Stil *bs_AutoRadioButton* initialisiert. Aufgrund dieser Stile kann nur ein Schaltfeld pro Gruppe ausgewählt werden. Wenn ein Feld eingeschaltet ist, sind die anderen automatisch ausgeschaltet.

Wenn Sie die Stile für Auswahl- und Schaltfelderobjekte als »nicht automatisch« neu definieren, müssen Sie selbst für den Status »ein« oder »aus« der Wahlfelder als Antwort auf das Anklicken sorgen. Sehen Sie sich dazu folgendes Beispiel an:

```
ChBox1^.Attr.Style := ChBox1^.Attr.Style and not
  bs_AutoCheckBox or bs_CheckBox
```

Informationen über den Status von Wahlfeldern

Die Abfrage eines Wahlfeldes ist eine Möglichkeit, seinen Status zu erfahren und darauf zu antworten. Schalt- und Auswahlfelder haben zwei »Zustände«:

ein- und ausgeschaltet. Mit der Methode *GetCheck* vom Typ *TCheckBox* können Sie den Status eines Wahlfeldes abfragen:

```
MyState := Check1^.GetCheck;
```

Der Rückgabewert von *GetCheck* kann mit den drei definierten Konstanten *bf_Unchecked*, *bf_Checked* und *bf_Grayed* zur Feststellung des Feld-Status verglichen werden.

Ändern des Wahlfeld-Status

Das Ändern (Ein- oder Ausschalten) des Wahlfeld-Status scheint die Aufgabe des Anwenders, nicht die Ihres Programms zu sein. Aber in einigen Fällen muß das Programm den Status eines Wahlfeldes direkt ändern können. Dies ist beispielsweise dann der Fall, wenn Sie Optionen anzeigen wollen, die im Vorhergehenden gewählt und gespeichert wurden. Der Typ *TCheckBox* definiert vier Methoden für das Ändern des Status eines Auswahlfeldes: *Check*, *Uncheck*, *Toggle* und *SetCheck* (die allgemeinste Methode).

Check bewirkt, daß der Status des Auswahlfeldes auf »ein« geschaltet wird:

```
Check1^.Check;
```

Uncheck schaltet den Status auf »aus«:

```
Check1^.Uncheck;
```

Toggle ändert den Status von »ein« auf »aus« oder umgekehrt. Im Fall von dreistufigen Schaltern ändert *Toggle* ein ausgeschaltetes Feld auf »ein«, ein eingeschaltetes Feld auf »deaktiviert« und ein deaktiviertes Feld auf »aus« um.

```
CheckBox1^.Toggle;
```

Mit *SetCheck* haben Sie die volle Kontrolle über den Status des Auswahlfeldes:

```
Check1^.SetCheck(0);                { Ausschalten }
Check1^.SetCheck(1);                { Einschalten }
```

Wenn Sie diese Methoden auf Schaltfelder anwenden, stellt ObjectWindows sicher, daß nur ein Schalter pro Gruppe auf »ein« steht. Dies gilt dann, wenn die Schalter zu einer Gruppe gehören.

Gruppen

In ihrer einfachsten Form ist eine Gruppe ein beschriftetes statisches Rechteck, das andere Steuerelemente optisch gruppiert. Der *Init*-Konstruktor einer Gruppe benötigt ein übergeordnetes Fenster, eine ID, einen Textstring, Positionierung und Größe als Argumente:

```
GroupBox1 := New(PGroupBox, Init(@Self, id_GB1, 'A Group Box',
  38, 102, 176, 108));
```

Zusätzlich zur optischen Gruppierung von Steuerelementen kann eine Gruppe auch zur logischen Bindung von Wahlfeldern (Auswahl- und Schaltfelder) verwendet werden. Diese logische Gruppe führt das automatische Ausschalten für Wahlfelder mit dem Stil »auto« aus.

Um einer Gruppe ein Wahlfeld hinzuzufügen, wird bei dessen Konstruktion ein Zeiger auf eine Gruppe definiert.

Beantworten von Gruppenbotschaften

Wenn ein Ereignis eingetreten ist, das die Auswahlpunkte in einer Gruppe geändert hat (beispielsweise wenn ein Anwender einen Schalter betätigt oder das Programm *Check* aufruft), empfängt das der Gruppe übergeordnete Fenster eine Botschaft, die auf der ID des untergeordneten Fensters basiert. Das übergeordnete Fenster fängt die Botschaft unter Verwendung der Summe von *id_First* und der ID der Gruppe ab. Dadurch können Sie Methoden für jede Gruppe anstatt für jedes Wahlfeld in der Gruppe definieren.

Um herauszufinden, welches Steuerelement in der Gruppe betroffen ist, können Sie den Status der Steuerelemente heranziehen.

Beispielprogramm: BtnTest

BtnTest ist ein vollständiges Programm, das ein Fenster mit Aktionsschalter-, Auswahlfelder-, Schaltfelder- und Gruppen-Steuerelementen erstellt. Beim Start der Anwendung wird das Hauptfenster mit den Steuerelementen angezeigt. Wenn der Anwender diese anklickt, gibt die Anwendung die entsprechenden unterschiedlichen Antworten (vgl. Abbildung 12.6).

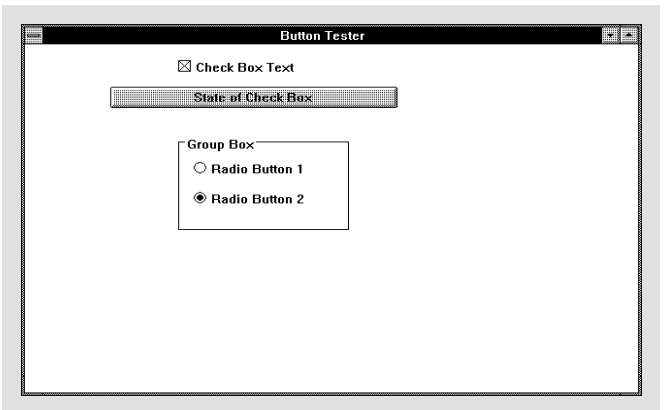


Abbildung 12.6 Ein Fenster mit verschiedenen Schaltern

Das vollständige Programm finden Sie unter dem Namen BTNTEST.PAS auf Ihren Originaldisketten.

Bildlaufleisten

Bildlaufleisten sind die wichtigsten Hilfsmittel, mit denen der Anwender die Anzeige eines Anwendungsfensters, einer Liste oder eines Kombinationsfensters verschieben kann. Es gibt aber auch Fälle, in denen Sie eine Bildlaufleiste zur Ausführung einer speziellen Aufgabe einsetzen möchten, wie zum Beispiel zur Steuerung der Temperatur in einem Thermostat-Programm oder der Farbe in einem Zeichen-Programm. Verwenden Sie *TScrollBar*-Objekte, wenn Sie eine separate, frei definierbare Bildlaufleiste benötigen.

In Abbildung 12.7 sehen Sie den typischen Gebrauch eines *TScrollBar*-Objekts.

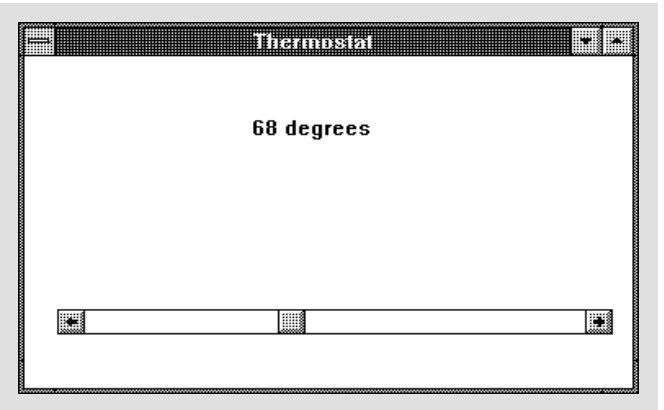


Abbildung 12.7
Ein Bildlaufleistenobjekt

Konstruieren von Bildlaufleisten

Der *Init*-Konstruktor der Bildlaufleiste benötigt ein übergeordnetes Fenster, eine ID, eine Position (*X* und *Y*), Breite, Höhe und ein Boolsches Flag (das festlegt, ob es sich um eine vertikale oder horizontale Leiste handelt) als Argumente. Ist für eine vertikale Bildlaufleiste eine Breite von 0 angegeben, wird diese mit einer Standardbreite konstruiert – wie die einer Liste. Dasselbe gilt, wenn die Höhe einer horizontalen Leiste mit 0 angegeben ist. Der Code:

```
ThermScroll := New(PScrollBar, Init(@Self, id_ThermScroll, 20,
  170, 340, 0, True));
```

erstellt die horizontale Bildlaufleiste in der Standardhöhe (vgl. Abbildung 12.7). *Init* konstruiert Bildlaufleisten mit den Stilen *ws_Child*, *ws_Visible* und *sbs_Horz* oder *sbs_Vert* für horizontale bzw. vertikale Bildlaufleisten. Sie können weitere Stile hinzufügen, wie zum Beispiel *sbs_TopAlign*, indem Sie das *Attr.Style*-Feld der Bildlaufleiste ändern.

In der Abbildung 12.8 sehen Sie verschiedene Bildlaufleisten.

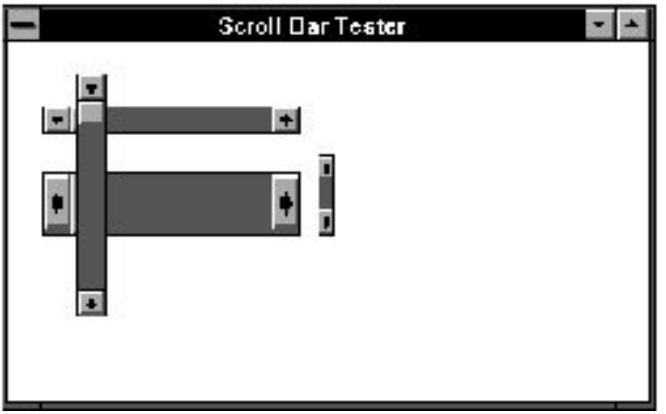


Abbildung 12.8
Ein Fenster mit verschiedenen Bildlaufleisten

Ein Attribut des Bildlaufleistenobjekts, das zur Konstruktion initialisiert wird, ist *range* (der Bereich). Der Bereich einer Bildlaufleiste ist die Menge aller möglichen Positionen des Bildlaufsymbols. Ein Bildlaufsymbol ist das verschiebbare Symbol, das der Anwender verschieben oder rollen kann. Jede Position ist mit einem Integerwert verknüpft. Das übergeordnete Fenster verwendet diesen Integerwert, d.h. die *Position*, um die Bildlaufleiste zu setzen und abzufragen. Bei der Konstruktion des Bildlaufleistenobjekts wird der Bereich auf 1 bis 100 festgelegt. Die »höchste« Position (der oberste Punkt bei vertikalen und der am weitesten links liegende Punkt bei horizontalen Bildlaufleisten) entspricht der Position 1. Entsprechend ist die »niedrigste« Position des Bildlaufsymbols auf 100 gesetzt. Mit der Methode *SetRange* (die weiter hinten in diesem Kapitel im Abschnitt »Ändern von Bildlaufleisten« erläutert wird) können Sie andere Werte für den Bereich angeben.

Zwei weitere Attribute für Bildlaufleisten sind die Zeilen- und Seitengrößen. Die Zeilengröße, initialisiert mit 1, ist die in Bereichseinheiten ausgedrückte Schrittweite, um die sich das Bildlaufsymbol verschiebt, wenn der Anwender die Pfeile der Bildlaufleiste anklickt. Die Seitengröße, initialisiert mit 10, ist die in Bereichseinheiten ausgedrückte Schrittweite, um die sich das Bildlaufsymbol verschiebt, wenn der Anwender im Rollbereich klickt. Sie können diese Werte neu setzen, indem Sie direkt die Objektfelder *LineSize* und *PageSize* von *TScrollBar* ändern.

Informationen über Bildlaufleisten

TScrollBar definiert zwei Methoden, die Informationen über Bildlaufleisten liefern: *GetRange* und *GetPosition*. Die *GetRange*-Methode ist eine Prozedur mit zwei Variablenargumenten vom Typ Integer. Die Prozedur speichert in diesen Variablen die höchste und niedrigste Position des Bildlaufsymbols im Bildlaufleistenbereich. Diese Methode ist hilfreich, wenn Ihr Programm das Bildlaufsymbol an die höchste oder niedrigste Position verschieben will.

GetPosition ist eine Funktion, die die Position des Bildlaufsymbols als Integer-

wert zurückliefert. Ihr Programm wird häufig Bereich und Position ermitteln und miteinander vergleichen.

Ändern von Bildlaufleisten

Das Ändern der Bildlaufleisten liegt in den meisten Fällen in der Hand des Anwenders. Aber auch Ihr Programm kann eine Bildlaufleiste ändern.

SetRange ist eine Prozedur, die zwei Integerargumente für die höchste und niedrigste Position im Bereich benötigt. Per Voreinstellung liegt der Bereich für neue Bildlaufleisten zwischen 1 und 100. Sie können diesen Bereich aber neu definieren, um eine bessere Anpassung an die tatsächlich verwendeten Einheiten der Bildlaufleiste zu erzielen. Die Bildlaufleisten in einer Thermostat-Anwendung sollten z.B. sinnvollerweise im Bereich von 32 bis 120 Grad Celsius liegen. Rufen Sie *SetRange* nach der Konstruktion des Bildlaufleistenobjekts auf:

```
procedure TParentWindowType.SetupWindow;
begin
  TWindow.SetupWindow;
  ThermScroll^.SetRange(32, 120);
end;
```

SetPosition ist eine Prozedur mit einem Integerargument für die Position, an die Sie das Bildlaufsymbol verschieben wollen. In der im vorhergehenden angesprochenen Thermostat-Anwendung könnte Ihr Programm die Temperatureinstellung beispielsweise direkt auf 78 Grad setzen:

```
ThermScroll^.SetPosition(78);
```

Die dritte Methode, *DeltaPos*, verschiebt die Position des Bildlaufsymbols um den im Integerargument angegebenen Umfang nach oben (bzw. nach links) oder unten (bzw. nach rechts). Eine positive Zahl bewegt das Bildlaufsymbol nach unten (rechts). Eine negative Zahl bewirkt ein Verschieben nach oben (links). Wenn Sie zum Beispiel die Temperatureinstellung um 5 Grad verringern wollen, verwenden Sie:

```
ThermScroll^.DeltaPos(-5);
```

Beantworten von Bildlaufleisten-Ereignissen

Beim Rollen einer Bildlaufleiste empfängt das übergeordnete Fenster eine entsprechende Benachrichtigungsbotschaft. Wenn Ihr Fenster auf Rollereignisse antworten soll, legen Sie auf die bekannte Weise Antwortmethoden an, die auf der ID des untergeordneten Fensters basieren. Benachrichtigungsbotschaften von Bildlaufleisten unterscheiden sich jedoch etwas von den übrigen Benachrichtigungsbotschaften von Steuerelementen. Sie basieren auf den Windows-Botschaften *wm_HScroll* und *wm_VScroll* und nicht auf *wm_Command*. Der

einzige Unterschied, auf den Sie achten müssen, liegt darin, daß die Benachrichtigungscodes von Bildlaufleisten in *Msg.wParam* und nicht in *Msg.lParamHi* gespeichert werden.

Zu den am meisten verwendeten Benachrichtigungscodes gehören: *sb_LineUp*, *sb_LineDown*, *sb_PageUp*, *sb_PageDown*, *sb_ThumbPosition* und *sb_ThumbTrack*. Oft wird die Reaktion auf alle Ereignisse im Überprüfen der neuen Position des Bildlaufsymbols und der entsprechenden Antwort beruhen. In diesem Fall können Sie den Benachrichtigungscode ignorieren. Sehen Sie sich dazu das folgende Beispiel an:

```
procedure TestWindow.HandleThermScrollMsg(var Msg: TMessage);
var
  NewPos: Integer;
begin
  NewPos := ThermScroll^.GetPosition;
  { Hier findet die Bearbeitung in Abhängigkeit
    von der neuen Position in NewPos statt }
end;
```

Eine häufige Alternative besteht darin, erst dann auf das Verschieben des Bildlaufsymbols zu antworten, wenn der Anwender eine neue Position ausgewählt hat.

Blenden Sie hierzu die *sb_ThumbTrack*-Botschaften aus:

```
procedure TestWindow.HandleThermScrollMsg(var Msg: TMessage);
var
  NewPos: Integer;
begin
  if Msg.wParam <> sb_ThumbTrack
  then begin
    newPos := ThermScroll^.GetPosition;
  { Hier findet die Bearbeitung in Abhängigkeit
    von der neuen Position in NewPos statt }
  end;
end;
```

Eventuell soll unter bestimmten Umständen das Bildlaufleistenobjekt selbst auf die Benachrichtigungsbotschaft antworten. Bauen Sie zu diesem Zweck ein spezielles Antwortverhalten in das Bildlaufleistenobjekt ein. Dazu definieren Sie für den Objekttyp eine benachrichtigungsbasierte Antwortmethode. Verwenden Sie die Summe von *nf_First* und den Benachrichtigungscode als Bezeichner im Methodenkopf.

Leiten Sie hierzu einen neuen Typ von *TScrollBar* ab:

```
SpecializedSBar = object(TScrollBar)
  procedure SBTop(var Msg: TMessage); virtual nf_First +
    sb_Top;
end;

procedure TSpecializedSBar.SBTop(var Msg: TMessage);
begin
```

```
    TScrollBar.SBTop(Msg);
    { Spezialbehandlung, wenn das Bildlaufsymbol
      auf den Anfang positioniert wird }
    Msg.Result := 1;
end;
```

Wenn das der Bildlaufleiste übergeordnete Fenster die Benachrichtigungsbotschaft empfangen soll, nachdem sie von der Bildlaufleiste bearbeitet wurde, muß diese die Botschaft explizit an das übergeordnete Fenster weitergeben. Dazu muß lediglich die Routine *DefNotificationProc* aufgerufen werden.

Beachten Sie, daß die Methode *TSpecializedSBar.SBTop* als erstes *TScrollBar.SBTop* aufruft. Dabei handelt es sich um die Methode, die sich um die physikalische Positionsänderung des Bildlaufsymbols kümmert.

TScrollBar bearbeitet die Botschaften (*nf_First* +) *sb_LineUp*, *sb_LineDown*, *sb_PageUp*, *sb_PageDown*, *sb_ThumbPosition*, *sb_ThumbTrack*, *sb_Top* und *sb_Bottom* mit den Methoden *SBLineUp*, *SBLineDown*, *SBPageUp*, *SBPageDown*, *SBThumbPosition*, *SBThumbTrack*, *SBTop* bzw. *SBBottom*. Wenn Sie weitere Verhaltensweisen und Eigenschaften hinzufügen wollen, rufen Sie zuerst die spezialisierte geerbte Bildlaufleisten-Methode auf. Andernfalls ist Ihre Methode für das Verschieben der Bildlaufleiste zuständig.

Beispiel: SBarTest

Das Programm *SBarTest* erstellt die Thermostat-Anwendung, die Sie in der Abbildung 12.7 sehen.

Sie finden das vollständige Programm unter dem Namen SBARTEST.PAS auf Ihren Originaldisketten.

Editierfelder

Editierfelder lassen sich als interaktive statische Steuerelemente beschreiben. Sie sind rechteckige Bereiche (mit und ohne Rahmen) auf dem Bildschirm, die vom Anwender oder dem Programm mit Text ausgefüllt, geändert und gelöscht werden können. Editierfelder sind vor allem als Felder für Dateneingabe-Bildschirme geeignet.

Diese Felder unterstützen folgende Operationen:

- Anwender-Texteingabe;
- Dynamische Textanzeige durch die Anwendung;
- Kopieren und Verschieben von und in die Zwischenablage;
- Mehrzeiliges Editieren (hilfreich für Texteditoren).

Die Abbildung 12.9 zeigt ein Fenster mit Editierfeldern.

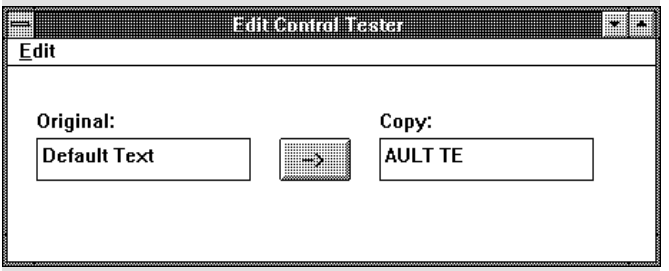


Abbildung 12.9
Ein Fenster mit Editierfeldern

Konstruieren von Editierfeldern

Der *Init*-Konstruktor von Editierfeldern benötigt ein übergeordnetes Fenster, eine Steuerelement-ID, *X*-, *Y*-, *W*- und *H*-Maßangaben, eine maximale Textlänge und das Boolesche Flag *Multiline* als Argumente. Das Editierfeld wird mit den Stilen *ws_Child* und *ws_Visible* (von *TControl.Init*), *ws_TabStop*, *es_Left* und *es_AutoHScroll* konstruiert.

Der Parameter für die Textlänge ist in Wirklichkeit um 1 größer als die maximale Anzahl der Zeichen, die in einer Editierzeile stehen dürfen, denn das Steuerelement benötigt eine 0 als abschließendes Zeichen. Wenn *Multiline* den Wert *False* hat, ist das Editierfeld eine einzige Zeile und erhält den Stil *ws_Border*.

```
EC1 := New(PEdit, Init(@Self, id_EC1, 'Default Text', 20, 50,
  150, 30, 40, False));
```

Wenn *Multiline True* ist, erhält das Editierfeld die Stile *es_Multiline*, *es_AutoVScroll*, *ws_VScroll* und *ws_HScroll*:

```
EC2 := New(PEdit, Init(@Self, id_EC2, '', 20, 20, 200, 150, 40,
  True));
```

Nach dem Aufruf von *Init* kann der Stil des Editierfeldes geändert werden:

```
EC3 := New(PEdit, Init(@Self, id_EC3, 'Default Text', 20, 20,
  150, 20, 40, False));
EC3^.Attr.Style := EC3^.Attr.Style and (not ws_Border);
```

Diese Zeilen produzieren das Editierfeld in Abbildung 12.10.

In der folgenden Tabelle 12.3 sind einige der gebräuchlichsten Stile für Editierfelder zusammengestellt.

Tabelle 12.3 Gebräuchliche Stile für Editierfelder

Stil	Ergebnis
es_Left	Ausrichtung links
es_Center	Zentriert
es_Right	Ausrichtung rechts
es_Multiline	Mehrzeiliges Editierfeld

Stil	Ergebnis
es_UpperCase	Umwandlung in Großbuchstaben
es_LowerCase	Umwandlung in Kleinbuchstaben
es_AutoVScroll	Hinzufügen von automatischem vertikalen Rollen bei mehrzeiligen Editierfeldern
es_AutoHScroll	Hinzufügen von automatischem horizontalen Rollen
ws_Border	Anbringen eines rechteckigen Rahmens um das Editierfeld
ws_VScroll	Hinzufügen einer vertikalen Bildlaufleiste
ws_HScroll	Hinzufügen einer horizontalen Bildlaufleiste

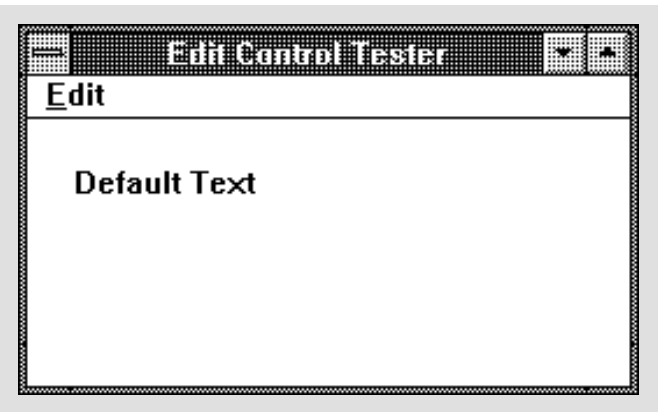


Abbildung 12.10 Ein Editierfeld ohne Rahmen

Zwischenablage und Editieroperationen

Sie können Text direkt von einem Editierfeld in die Zwischenablage von Windows bringen und umgekehrt. Die Methoden für die Zwischenablage werden folgendermaßen aufgerufen:

```
EC1^.Cut
EC1^.Copy;
EC1^.Paste;
```

Außerdem stehen folgende Editiermethoden zur Verfügung:

```
EC1^.Clear;
EC1^.DeleteSelection;
EC1^.Undo;
```

Sie werden dem Anwender häufig über ein Editmenü den Zugriff auf diese Methoden ermöglichen. Ein Editierfeldobjekt kann unmitelbar auf die Auswahl von Menüpunkten wie *Edit/Copy* und *Edit/Undo* antworten. *TEdit* definiert auf Befehlen basierende Methoden, wie zum Beispiel *CMEditCopy* und *CMEditUndo*,

die als Antwort auf eine bestimmte Menüauswahl (einen Befehl) im übergeordneten Fenster des Editierfeldes aufgerufen werden. *CMEditCopy* ruft *Copy* und *CMEditUndo* ruft *Undo* auf. In der folgenden Tabelle sind die Methoden aufgeführt, die als Antwort auf Menüauswahlen mit einer bestimmten Menü-ID (definiert in WOBJECTS.H) aufgerufen werden:

Tabelle 12.4 Menü-IDs und die von ihnen aufgerufenen Methoden

Menü-ID	Aufgerufene Methode
cm_EditCut	*Cut*
cm_EditCopy	*Copy*
cm_EditPaste	*Paste*
cm_EditClear	*Clear*
cm_EditDelete	*DeleteSelection*
cm_EditUndo	*Undo*

Wenn Sie ein Editmenü in ein Fenster einfügen wollen, das über Editierfeldobjekte als untergeordnete Fenster verfügt, müssen Sie nur eine Menüressource für das Fenster definieren, die die in der Ressource aufgeführten Menü-IDs verwendet. Sie müssen keine Methoden schreiben.

Eine weitere Editiermethode ist die Boolesche Funktion *CanUndo*, die feststellt, ob die zuletzt ausgeführte Änderung rückgängig gemacht werden kann.

Implementierung

Die Antwort des Editmenüs wird durch den ObjectWindows-Mechanismus erleichtert, der Botschaften an untergeordnete Fenster übergibt. Dieser Mechanismus gibt eine Botschaft, die von einem Ereignis in einem untergeordneten Fenster erzeugt wird, zuerst an das untergeordnete Fenster weiter. Wenn Steuerelemente untergeordnete Fenster sind, fängt das Steuerelement nur selten die Botschaft ab. Die Botschaft wird dann vom übergeordneten Fenster bearbeitet. Im Fall von Befehlsbotschaften, die vom Editmenü erzeugt werden, geht die Botschaft an das Editierfeld, das den Fokus hat.

Informationen über Editierfelder

Sie werden ein Editierfeld immer dann abfragen, wenn Sie den Eingabetext auf seine Gültigkeit überprüfen müssen oder wenn die Eingabe für eine spätere Verwendung gespeichert oder in ein anderes Steuerelement kopiert werden soll. *TEdit* unterstützt eine Vielzahl von Methoden, die der Abfrage und Information dienen. Viele Abfrage- und Änderungsmethoden liefern eine Zeilennummer oder Position eines Zeichens in der Zeile zurück bzw. erfordern deren Angabe. Alle Indexpositionen beginnen bei 0. Mit anderen Worten: Die erste Zeile ist die Zeile 0 und das erste Zeichen in einer Zeile ist das Zeichen 0. Die wichtigsten Abfragemethoden lauten: *GetText*, *GetLine*, *NumLines* und *LineLength*.

Die Abfragemethoden von *TEdit*, die Text aus einem Steuerelement zurück-

liefern, erhalten auch die Textformatierung. Dies spielt nur bei mehrzeiligen Editierfeldern eine Rolle, in denen Text in mehr als einer Zeile angezeigt wird. In diesem Fall enthält der Text zwei zusätzliche Zeichen für jeden erforderlichen Zeilenumbruch: eine *Carriage Return* (#13) und ein *Linefeed* (#10). Wenn der Text wieder aus der Zwischenablage zurück in ein Editierfeld gebracht, in eine Datei geschrieben oder auf einem Drucker ausgegeben wird, erscheinen die Zeilenumbrüche genauso wie im Editierfeld. Deshalb müssen Sie bei der Verwendung einer Abfragemethode, die eine Zeichenanzahl ermittelt, immer die beiden zusätzlichen Zeichen für den Zeilenumbruch berücksichtigen.

GetText holt den Text aus einem Editierfeld und füllt den String, auf den das übergebene PChar-Argument zeigt, mit dem Inhalt des Editierfeldes mit der Anzahl von Zeichen aus, die im zweiten Parameter festgelegt sind, einschließlich der Zeilenumbrüche. Der Rückgabewert ist *False*, wenn das Editierfeld leer ist oder wenn in ihm mehr Text steht, als im vorgesehenen String Platz findet. Die folgende Prozedur holt Text aus einem Editierfeld und liefert ihn im *RetText*-Argument zurück:

```
procedure TTestWindow.ReturnText(RetText: PChar);
var TheText: array[0..20] of Char;
begin
  if EC1^.GetText(@TheText, 20) then
    RetText := @TheText
  else RetText := nil;
end;
```

GetLine hat eine speziellere Wirkung als *GetText*. Es liefert in einem mehrzeiligen Editierfeld den Text aus der Zeile, die in dem Integerargument angegeben wird. Zeile 0 ist die erste Zeile.

NumLines liefert die Anzahl der Zeilen in einem mehrzeiligen Editierfeld. Sie sollten *NumLines* verwenden, wenn Sie überprüfen wollen, wieviele Zeilen in ein Editierfeld eingegeben wurden, bevor Sie Text mit *GetLine* aus einer Zeile holen.

LineLength liefert die Anzahl der Zeichen in der angegebenen Zeile eines mehrzeiligen Editierfeldes. Wenn die Zeile existiert, aber keine Zeichen enthält, liefert *LineLength* 0 zurück. *LineLength* sollte zusammen mit *NumLines* und *GetLine* verwendet werden, um Text aus einem mehrzeiligen Editierfeld zu holen:

```
procedure TestWindow.ReturnLineText(RetText: PChar; LineNum:
  Integer);
var
  TheText: array[0..20] of Char;
begin
  RetTExt := nil;
  if EC1^.NumLines >= LineNum then
    if EC1^.LineLength(LineNum) < 11 then
      if EC1^.GetLine(@TheText, 20, LineNum) then
        RetText := @TheText;
end;
```

GetSubText ist noch spezialisierter als *GetText*. Es benötigt ein PChar- und zwei Integerwerte als Argumente, die den Start- und Endwert eines Textbereichs in einem Editierfeld festlegen. Das erste Zeichen hat die Position 0. *GetSubText* liefert den String mit den Zeichen, die zwischen den beiden Indexpositionen liegen. In einem mehrzeiligen Editierfeld wird der Index sequentiell von der ersten Zeile an hochgezählt. Zeilenumbrüche werden als zwei Zeichen gezählt. Dadurch kann Ihr Programm das Format des Textes im Editierfeld rekonstruieren, wenn der Text angezeigt, gedruckt, in ein anderes Editierfeld übertragen oder in die Zwischenablage gebracht wird.

Sie werden sich nun fragen, woher die Indizes kommen, die als Argumente in den Aufruf von *GetSubText* übergeben werden. Eine Möglichkeit ist die Verwendung von *GetSelection*, einer Methode, die Start- und Endwert des aktuell gewählten (hervorgehobenen) Bereichs zurückliefert. Normalerweise wird dieser Bereich vom Anwender ausgewählt, dies kann aber auch vom Programm mit der Methode *SetSelection* durchgeführt werden. Vergleichen Sie dazu den folgenden Abschnitt »Ändern von Editierfeldern«. Die letzte Indexposition, die von *GetSelection* zurückgeliefert wird, ist die Position des letzten ausgewählten Zeichens plus 1.

Sie können in einer Anwendung eventuell Zeit sparen, die für die Bearbeitung von Text aus Editierfeldern erforderlich ist. Die *IsModified*-Methode, eine Boolesche Funktion, liefert den Wert *True*, wenn der Anwender den Text in einem Editierfeld geändert hat, und *False*, wenn der Text unverändert ist. Die Rückgabe von *False* kann Ihnen den Aufruf von *GetText* und *GetLine* ersparen. *ClearModify* setzt das Flag wieder auf *False*.

LineIndex und *GetLineFromPos* sind zwei Methoden, mit deren Hilfe Sie die Position von Text in einem mehrzeiligen Editierfeld berechnen können. *LineIndex* liefert die Anzahl der Zeichen (einschließlich der beiden Zeichen für Zeilenumbrüche) in allen Zeilen, die vor der im Integerargument angegebenen Zeile liegen. Die Methode liefert alle Zeichen des Editierfeldes, wenn die angegebene Zeile nicht existiert. *GetLineFromPos* benötigt einen Index für eine Zeichenposition und liefert die Zeilennummer, die zu dieser Position gehört. Beide Methoden sind dann hilfreich, wenn Sie die Struktur der Zeilen und die Inhalte eines mehrzeiligen Editierfeldes abfragen wollen.

Ändern von Editierfeldern

In einem herkömmlichen Programm zur Dateneingabe besteht für das Programm keine Veranlassung, ein Editierfeld direkt zu ändern. Der Anwender ändert den Text und das Programm liest den Text mit einer Abfrage-Methode. Da Editierfelder auch für viele andere Zwecke verwendet werden, muß auch die Anwendung selbst Text ersetzen, einfügen, löschen oder auswählen können. ObjectWindows unterstützt dieses Verhalten und kann außerdem die Editierfelder rollen.

Text löschen

Clear löscht einfach den gesamten Text in einem Editierfeld. Sie können *Clear*

als Teil einer Initialisierung verwenden, die einige oder alle Editierfelder in einem Fenster löscht. *DeleteSelection* löscht den aktuell ausgewählten Text eines Editierfeldes. Ein Texteditor kann den Zugriff auf *Clear* und *DeleteSelection* über Menüauswahlpunkte zur Verfügung stellen (vgl. »Zwischenablage und Editieroperationen« weiter vorne in diesem Kapitel). *DeleteSelection* ist eine Boolesche Funktion, die *False* zurückliefert, wenn kein Text ausgewählt ist.

DeleteSubText ähnelt *DeleteSelection*. Es löscht jedoch den Text, der zwischen der übergebenen Anfangs- und Endposition steht. Der Text wird inklusive des ersten, aber exklusive des letzten Zeichens gelöscht.

DeleteLine löscht den gesamten Text, der in der abgegebenen Zeile steht – jedoch nicht die zwei Zeichen für den Zeilenumbruch. Alle anderen Zeilen bleiben unverändert.

Text einfügen

Insert hat eine ähnliche Funktion wie *Paste*. *Insert* holt den Text, der eingefügt werden soll, nicht aus der Zwischenablage, sondern aus dem übergebenen Argument. Es löscht vorhandenen ausgewählten Text, hebt jedoch den eingefügten Text optisch nicht hervor. Mit *Insert* können Sie Ihre eigene Zwischenablage (nur für Text) im Programm implementieren.

SetText kombiniert die Aktionen von *Clear* und *Insert*. Es löscht den gesamten Inhalt des Editierfeldes und fügt den Text in das übergebene Argument ein. *SetText* kann aus diesem Grund für Initialisierungsoperationen verwendet werden. Wenn Sie beispielsweise einen Bestell-Eingabebildschirm für neue Bestellungen initialisieren möchten, die in erster Linie aus Kalifornien kommen, könnten Sie *SetText* wie folgt aufrufen:

```
StateField^.SetText('CA');
```

und damit das *state*-Eingabefeld setzen.

Textauswahl und Rollen

SelectRange bewirkt die Auswahl oder Markierung (optische Hervorhebung) von Text zwischen der übergebenen Anfangs- und Endposition, wobei jedoch das Zeichen an der Endposition nicht eingeschlossen wird.

Scroll bewirkt das Rollen eines mehrzeiligen Editierfeldes. *Scroll* benötigt zwei Integerargumente: die Anzahl der Zeichen, um die horizontal verschoben wird, und die Anzahl der Zeilen, um die vertikal gerollt wird. Eine positive Integerzahl rollt nach rechts bzw. nach unten, eine negative Integerzahl nach links bzw. nach oben. Vertikal können nur mehrzeilige Editierfelder gerollt werden.

Beispielprogramm: EditTest

EditTest ist ein Programm mit einem Hauptfenster, das als übergeordnetes Fenster für zwei Editierfelder, zwei statische Steuerelemente und einen Schalter dient. Abbildung 12.9 stellt dieses Fenster dar. Wenn der Anwender den Schalter anklickt, wird der Text aus dem linken Editierfeld (*EC1*) in das rechte (*EC2*) kopiert. Der Text wird in *EC2* in Großbuchstaben umgewandelt, weil dieses Feld

mit dem Stil *es_UpperCase* konstruiert wurde. Ist in *EC1* kein Textbereich ausgewählt, wird der gesamte Text in *EC2* kopiert. Ist in *EC1* Text ausgewählt, wird nur dieser Teil kopiert.

Das Editmenü unterstützt Editierfunktionen für das Editierfeld, das gerade den Fokus hat.

Das vollständige Programm finden Sie unter dem Namen EDITTEST.PAS und seine Ressourcendatei unter dem Namen EDITTEST.RES auf Ihren Originaldisketten.

Kombinationsfenster

Ein Kombinationsfenster ist eine Kombination aus zwei anderen Steuerelementen: einer Liste und einem Editierfeld. Es dient zu denselben Zwecken wie eine Liste – der Anwender kann einen Eintrag aus einer rollbaren Liste mit Texteinträgen auswählen, indem er den gewünschten Eintrag mit der Maus anklickt. Das Editierfeld, das oberhalb der Liste liegt, stellt einen weiteren Auswahlmechanismus zur Verfügung. Hier kann der Anwender den Text des gewünschten Eintrags eintippen. Falls der Listenbereich des Kombinationsfensters angezeigt ist, wird der gewünschte Eintrag automatisch ausgewählt.

Der Typ *TComboBox* stammt vom Typ *TListBox* ab und erbt dessen Methoden für das Ändern, Abfragen und Auswählen von Listeneinträgen. *TComboBox* verfügt über zusätzliche Methoden zur Manipulation des Listenteils im Kombinationsfenster, der in bestimmten Fällen auf Anforderung »herunterfallend« (*Drop-down*-Liste) aufspringt.

Drei Arten von Kombinationsfenstern

Es gibt drei Arten von Kombinationsfenstern: einfache, aufspringende und aufspringende Listen-Kombinationsfenster. Abbildung 12.11 zeigt diese drei Arten und eine Liste.

Einfache Kombinationsfenster

Alle Kombinationsfenster zeigen immer ihren Editierbereich an, manche Kombinationsfenster können jedoch den Listenbereich ein- oder ausblenden. Ein einfaches Kombinationsfenster kann dies nicht, der Listenbereich wird immer angezeigt. Der Editierbereich verhält sich wie ein Editierfeld. Der Anwender kann Text eingeben, wobei der eingetippte Text nicht einem Eintrag in der Liste entsprechen muß. Ist das jedoch der Fall, wird der entsprechende Eintrag in der Liste ausgewählt.

Aufspringende Kombinationsfenster

Ein aufspringendes Kombinationsfenster entspricht einem einfachen Kombinationsfenster mit einer einzigen Ausnahme: Zu Beginn ist der Listenbereich

nicht eingeblendet. Dieser Bereich wird erst dann angezeigt, wenn der Anwender den Pfeil nach unten auf der rechten Seite des Editierbereichs anklickt. Aufspringende Kombinationsfenster sind sehr nützlich, wenn viele Steuerelemente in einen kleinen Bereich passen sollen.

Wenn die aufspringenden Listen nicht benötigt werden, belegen sie viel weniger Platz als einfache Kombinationsfenster oder normale Listen.

Aufspringende Listen-Kombinationsfenster

Der Listenbereich eines derartigen Kombinationsfensters verhält sich wie der Listenbereich in einem aufspringenden Kombinationsfenster – er wird nur dann eingeblendet, wenn er benötigt wird, und danach wieder ausgeblendet.

Diese beiden aufspringenden Arten unterscheiden sich im Verhalten ihrer Editierbereiche. Während sich die Editierbereiche der aufspringenden Kombinationsfenster wie reguläre Editierfelder verhalten, können die Editierbereiche in aufspringenden Listen-Kombinationsfenstern nur den Text eines einzigen Listeneintrags anzeigen. Wenn der Text im Editierbereich mit dem Text eines Eintrags übereinstimmt, können keine weiteren Zeichen eingegeben werden.

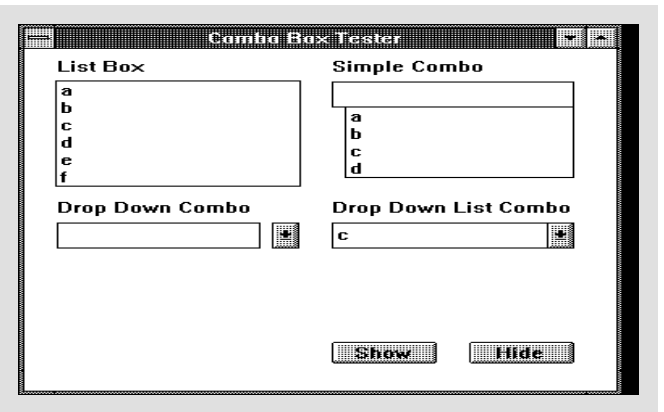


Abbildung 12.11 Die drei Arten von Kombinationsfenstern und eine Liste

Auswahl einer Kombinationsfenster-Art

Aufspringende Listen-Kombinationsfenster sind besonders für Fälle geeignet, in denen nur Einträge ausgewählt werden können, die in der Liste stehen.

So können zum Beispiel bei der Auswahl von Druckern nur diejenigen ausgewählt werden, auf die Ihr System zugreifen kann (vergleichen Sie hierzu Abbildung 12.12).

Aufspringende Kombinationsfenster können jedoch auch Einträge akzeptieren, die nicht in der Liste aufgeführt sind. Eine Einsatzmöglichkeit für diese Fenster ist die Auswahl von Dateien, die geöffnet oder gespeichert werden sollen. Der Anwender kann sowohl die Verzeichnisse nach der gewünschten Datei in der Liste durchsuchen als auch den vollen Pfad- und Dateinamen im

Editierbereich eingeben, unabhängig davon, ob der Dateiname im Listenbereich angezeigt wird. Sehen Sie sich dazu die Abbildung 12.13 an.

Abbildung 12.12 Ein aufspringendes Listen-Kombinationsfenster

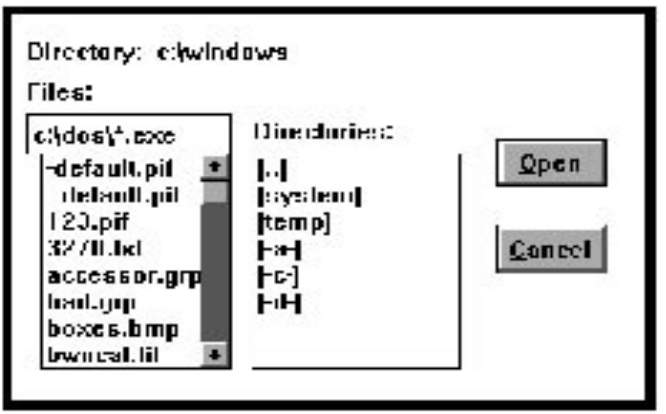


Abbildung 12.13 Ein einfaches Kombinationsfenster

Konstruieren von Kombinationsfenstern

Der *Init*-Konstruktor für *TComboBox* benötigt ein übergeordnetes Fenster, eine ID, *X*, *Y*, *W* und *H*, einen Stil und die maximale Länge als Argumente. Alle mit *Init* konstruierten Kombinationsfenster haben die Stile *ws_Child*, *ws_Visible*, *cbs_AutoHScroll*, *cbs_Sort* (sortierte Liste) und *ws_VScroll* (vertikale Bildlaufleiste). Der Stilparameter ist einer der Standardstile von Windows für Kombinationsfenster: *cbs_Simple*, *cbs_DropDown* oder *cbs_DropDownList*. Der Parameter für die Textlänge hat dieselbe Wirkung wie der entsprechende Parameter für ein Editierfeld: Er beschränkt die Anzahl der Zeichen, die in den Editierbereich des Kombinationsfensters eingegeben werden können. Die folgenden Zeilen produzieren ein aufspringendes Listen-Kombinationsfenster mit einer nicht sortierten Liste:

```
CD3 := New(PComboBox, Init(@Self, id_CB3, 190, 160, 150, 100,
  cbs_DropDownList, 40));
CB3^. Attr.Style := CB3^.Attr.Style and (not cbs_Sort);
```

Ändern von Kombinationsfenstern

TComboBox definiert zwei Methoden, die den Listenbereich von aufspringenden Kombinationsfenstern ein- und ausblenden: *ShowList* und *HideList*. Beide Methoden sind Prozeduren und benötigen keine Argumente. Sie müssen diese Methoden zum Ein- und Ausblenden der Liste nicht aufrufen, wenn der Anwender den Pfeil nach unten rechts neben dem Editierbereich anklickt. Dieser Mechanismus wird in Kombinationsfenstern automatisch aktiviert. Diese Methoden sind nur geeignet, um das Ein- und Ausblenden der Liste zu erzwingen.

Beispielprogramm: CBoxTest

Das Programm *CBoxTest* produziert die in Abbildung 12.11 gezeigte Anwendung und verwendet alle drei Arten von Kombinationsfenstern. *CB1* ist ein einfaches, *CB2* ein aufspringendes und *CB3* ein aufspringendes Listen-Kombinationsfenster.

Das Anklicken der Schalter *Show* und *Hide* hat das Ein- und Ausblenden des rechten unteren Kombinationfensters *CB3* zur Folge. Zu diesem Zweck werden die Methoden *ShowList* und *HideList* aufgerufen. Das vollständige Programm finden Sie unter dem Namen CBOXTEST.PAS auf Ihren Originaldisketten.

Setzen von Werten für Steuerelemente

Zur Verwaltung von komplexen Dialogen und Fenstern mit vielen untergeordneten Steuerelementen werden Sie normalerweise einen Objekttyp ableiten, der den Status seiner Steuerelemente ermittelt und speichert. Der Status eines Steuerelements beinhaltet den Text eines Editierfeldes, die Position eines Bildlaufsymbols und die Information, ob ein Schaltfeld eingeschaltet ist. Sie können aber auch die Definition eines abgeleiteten Typs umgehen, indem Sie einen Record definieren, der den Status der Steuerelemente eines Fensters oder Dialogs repräsentiert. Dieser Record wird als Transferpuffer bezeichnet, weil die Statusinformation auf einfache Weise zwischen dem Puffer und den Steuerelementen übertragen werden kann.

Nehmen wir folgendes Beispiel an: Ihr Programm zeigt einen modalen Dialog an und extrahiert, nachdem der Dialog geschlossen wurde, Informationen über den Status der Steuerelemente aus dem Transferpuffer. Wenn der Anwender den Dialog wieder aktiviert, werden die Steuerelemente des Dialogs auf den Status

gesetzt, der beim letzten Schließen des Dialogs gültig war. Zusätzlich können Sie den anfänglichen Status jedes Steuerelements gemäß den Informationen im Transferpuffer bestimmen. Sie können auch jederzeit Informationen in beiden Richtungen explizit übertragen, um beispielsweise den Status von Steuerelementen mit den vorhergehenden Werten zu besetzen. Ein Fenster oder nichtmodaler Dialog mit Steuerelementen kann ebenfalls diesen Mechanismus verwenden, um jederzeit Statusinformationen zu setzen oder zu ermitteln.

Die Verknüpfung von Steuerelementobjekten mit Steuerelementen wird in Kapitel 11 »Dialogobjekte« erläutert.

Für den Transfermechanismus müssen Sie ObjectWindows-Objekte verwenden, die die Steuerelemente repräsentieren, für die Informationen übertragen werden sollen. Das bedeutet, daß Sie mit *InitResource* Objekte mit Steuerelementen in Dialogen oder Fenstern verknüpfen müssen.

Definieren eines Transferpuffers

Der Transferpuffer ist ein Record, der für jedes vom Transfervorgang betroffene Steuerelement ein Feld zur Verfügung stellt. Ein Fenster oder Dialog kann aber auch Steuerelemente enthalten, deren Werte nicht vom Transfermechanismus bestimmt werden. Zum Beispiel sind Aktionsschalter, die keinen Status haben, und Gruppen vom Transfermechanismus nicht betroffen.

Zur Definition eines Transferpuffers legen Sie für jedes vom Transfer betroffene Steuerelement im Dialog oder Fenster ein Feld an. Es ist aber nicht notwendig, für alle Steuerelemente Felder anzulegen. Felder werden nur für die Elemente benötigt, in die Werte übertragen oder aus denen Werte extrahiert werden sollen. Der folgende Transferpuffer speichert ein Steuerelement von jedem Typ, mit Ausnahme von Aktionsschaltern und Gruppen:

```
type
  SampleTransferRecord = record
    Stat1: array[0..TextLen-1] of Char; { statischer Text }
    Edit1: array[0..TextLen-1] of Char; { Text des Editierfeldes }
    List1Strings: PStrCollection;       { Listenstrings }
    List1Selection: Integer;            { Index des ausgewählten
                                          Strings }
    ComboStrings: PStrCollection;       { Strings des Kombinations-
                                          fensters }
    ComboSelection: array[0..TextLen-1] of Char;
                                        { ausgewählter String }
    Check1: Word;                       { Status des Auswahlfeldes }
    Radio1: Word;                       { Status des Schaltfeldes }
    Scroll1: ScrollBarTransferRec;      { Bereich der Bildlauf-
                                          leiste, etc. }
```

Wie Sie sehen, bestimmt der Typ des Steuerelements den Typ des Feldes, das für den Transferpuffer definiert wird. Denn jeder Steuerelementtyp muß unterschiedliche Informationen speichern:

- Statische Steuerelemente speichern ein Zeichen-Array bis zur maximal zulässigen Textlänge plus einer terminierenden 0.

- Editierfelder speichern Textpuffer des Editierfeldes bis zu der Länge, die im Feld *TextLen* festgelegt ist.

- Listen und Kombinationsfenster speichern die Kollektion der Strings in der Liste und eine Angabe des ausgewählten Eintrags bzw. der ausgewählten Einträge. Für eine Liste mit einer einzigen Auswahl wird ein Integerindex für den ausgewählten String gespeichert. Einer Liste mit mehreren Einträgen steht ein Record zur Verfügung, der die Indizes aller ausgewählten Einträge enthält. Für ein Kombinationsfenster wird der ausgewählte String selbst gespeichert.

- Der Status von Auswahl- und Schaltfelder wird als Wert vom Typ Word mit den Werten *bf_Unchecked*, *bf_Checked* und *bf_grayed* gespeichert.

- Bildlaufleisten verwenden einen Record mit dem Namen *ScrollBarTransferRec* zum Speichern des Bereichs und der Position des Bildlaufsymbols. *ScrollBarTransferRec* ist folgendermaßen definiert:

```
ScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
```

Definieren des entsprechenden Dialogs oder Fensters

Fenster oder Dialoge, die den Transfermechanismus verwenden, müssen die davon betroffenen Steuerelementobjekte genau in der Reihenfolge konstruieren, in der die entsprechenden Felder im Transferpuffer definiert sind. Zur Aktivierung des Transfermechanismus für ein Fenster- oder ein Dialogobjekt muß nur das Feld *TransferBuffer* des Objekts mit einem Zeiger auf den definierten Transferpuffer belegt werden.

Bei Fenstern mit Steuerelementen müssen die Steuerelementobjekte mit *Init* konstruiert werden. Für Dialogefenster muß der Konstruktor *InitResource* eingesetzt werden. Sehen Sie sich dazu folgendes Beispiel an:

```
type
  SampleTransferRecord = record
  ...
  PParentWindow = ^TParentWindow;
  TParentWindow = object(TWindow);
    TheDialog: PDialog;
    TheBuffer: SampleTransferRecord;

    ...

constructor TParentWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
```

```
var
  Stat1: PStatic;
  Edit1: PEdit;
  List1: PListBox;
  Combo1: PComboBox;
  Check1: PCheckBox;
  Radio1: PRadioButton;
  Scroll1: PScrollBar;
begin
  TWindow.Init(AParent, ATitle);
  TheDialog^.Init(@Self, PChar(101));
  New(Stat1, InitResource(TheDialog, id_Stat1));
  New(Edit1, InitResource(TheDialog, id_Edit1));
  New(List1, InitResource(TheDialog, id_List1));
  New(Combo1, InitResource(TheDialog, id_Combo1));
  New(Check1, InitResource(TheDialog, id_Check1));
  New(Radio1, InitResource(TheDialog, id_Radio1));
  New(Scroll1, InitResource(TheDialog, id_Scroll1));
  TheDialog^.TransferBuffer := @TheBuffer;
end;
```

Für ein Fenster mit Steuerelementen verwenden Sie *Init* anstelle von *InitResource*, um die Steuerelemente in der richtigen Reihenfolge zu konstruieren. Ein weiterer Unterschied zwischen Fenstern und Dialogen liegt darin, daß der Transfermechanismus bei Steuerelementen eines Fensters standardmäßig zu Beginn ausgeschaltet ist. Zur Aktivierung des Mechanismus rufen Sie *EnableTransfer* auf:

```
...
Edit1 := New(PEdit, Init(@Self, id_Edit1, '', 10, 10, 100, 30,
  40, False));
Edit1^.EnableTransfer;
...
```

Wenn ein Steuerelement explizit vom Transfer ausgeschlossen werden soll, rufen Sie nach dessen Konstruktion seine *DisableTransfer*-Methode auf.

Übertragen der Informationen

Nach der Fenster- oder Dialogerstellung (einschließlich modaler Dialoge) werden Informationen vom Transferpuffer zum Setzen der betroffenen Steuerelemente übertragen. Für modale Dialoge werden die Informationen standardmäßig nur dann von den Steuerelementen in den Transferpuffer übertragen, wenn der Dialog eine Befehlsbotschaft mit der Steuerelement-ID *id_OK* empfängt. Da diese ID normalerweise zurückgeliefert wird, wenn der Anwender den *OK*-Schalter zum Beenden des Dialogs drückt, aktualisiert der Dialog automatisch seinen Transferpuffer. Wird der Dialog dann erneut verwendet, werden

die Informationen im Puffer wieder an die Steuerelemente übertragen. Auf diese Weise bleiben Dialog und Puffer immer synchron.

Sie können jedoch jederzeit Informationen in beiden Richtungen übertragen, beispielsweise Informationen aus einem Steuerelement in ein Fenster oder einen nicht-modalen Dialog. Sie können auch den Status der Steuerelemente mit den Informationen aus dem Transferpuffer als Antwort auf das Anklicken eines Reset-Schalters wiederherstellen. Verwenden Sie die Methode *TransferData* mit der Konstante *tf_SetData*, um Information aus dem Puffer zu den Steuerelementen zu übertragen, und mit der Konstante *tf_GetData* für den Transfer in der anderen Richtung. Folgendermaßen könnten Sie *TransferData* zum Beispiel in der *Destroy*-Methode eines Fensterobjekts aufrufen:

```
procedure TSampleWindow.Destroy;
begin
  TransferData(tf_GetData);
  TWindow.Destroy;
end;
```

Transfer bei frei definierten Steuerelementen

Sie können auch die Art und Weise ändern, in der ein bestimmtes Steuerelement seine Informationen überträgt oder ein neu definiertes Steuerelement in den Transfermechanismus einbeziehen. In beiden Fällen müssen Sie nur eine *Transfer*-Methode für Ihr Steuerelement schreiben, das – wenn das Flag *tf_GetData* gesetzt ist – Information aus dem Steuerelement an die durch den übergebenen Zeiger festgelegte Position kopiert. Wenn das Flag *tf_SetData* gesetzt ist, kopieren Sie die Information an der vom Zeiger angegebenen Position in das Steuerelement. Im folgenden sehen Sie als Beispiel *TStatic.Transfer*:

```
function TStatic.Transfer(DataPtr: Pointer; TransferFlag:
  Word);
begin
  if TransferFlag = tf_GetData then
    GetText(DataPtr, TextLen)
  else if TransferFlag = tf_SetData then
    SetText(DataPtr);
  Transfer := TextLen;
end;
```

Diese Methode sollte immer die Anzahl der übertragenen Bytes zurückliefern.

Transfer-Beispiel

Das Hauptfenster des Programms *TranTest* produziert ein modales Dialogfenster mit Feldern, in die der Anwender Name und Adresse einträgt. Es wird

ein Transferpuffer eingesetzt, der diese Informationen speichert und in den Steuerelementen des Dialogs anzeigt, wenn der Dialog erneut verwendet wird. Beachten Sie, daß kein neuer Dialogobjekttyp definiert werden muß, der die Dialoginformationen setzt und ermittelt. Beachten Sie außerdem, daß die Informationen im Transferpuffer direkt manipuliert werden, damit das statische Steuerelement bei der ersten Anzeige des Dialogs »First Mailing Label« beinhaltet, während alle weiteren Anzeigen »Subsequent Mailing Label« lauten.

Das vollständige Programm finden Sie unter dem Namen TRANTEST.PAS auf Ihren Originaldisketten.

Steuerelemente und die Botschaft wm_Paint

Die meisten ObjectWindows-Objekte antworten auf *wm_Paint*-Botschaften durch Aufruf ihrer *Paint*-Methoden. Steuerelemente werden aber auf etwas andere Weise neu gezeichnet. In den meisten Fällen müssen Sie das Erscheinungsbild eines Steuerelements gar nicht ändern – soll die Farbe eines Steuerelements oder Steuerelementtyps geändert werden, können Sie Ihr Dialog-Objekt auf *wm_CtlColor*-Botschaften antworten lassen.

Für die Änderung des Erscheinungsbildes eines Steuerelementes haben Sie verschiedene Möglichkeiten. Viele Steuerelemente werden mit einem *_OwnerDraw*-Stil erzeugt. Hierdurch wird Windows veranlaßt, dem übergeordneten Fenster gegebenenfalls eine *wm_DrawItem*-Botschaft zu senden, so daß dieses das Steuerelement neu zeichnen kann. Falls dieses Verfahren unzureichend ist, definieren Sie am besten einen vollständig neuen Steuerelementtyp, der sich selber auf geeignete Weise neu zeichnet.

Sie können natürlich auch die *WMPaint*-Antwortmethode überschreiben, um ein bestimmtes Steuerelement anders neu zu zeichnen. Dies dürfte aber keine besonders geeignete Alternative sein.

13. MDI-Objekte

Die Multidokumentschnittstelle (MDI) ist ein Schnittstellenstandard für Windows-Anwendungen, in denen der Anwender gleichzeitig in mehreren geöffneten Dokumenten arbeiten kann. Unter einem Dokument verstehen wir in diesem Zusammenhang eine dateispezifische Aufgabe wie das Editieren einer Textdatei oder die Arbeit in einer Kalkulationsdatei. In MDI-Anwendungen kann der Anwender viele geöffnete Dateien innerhalb einer Anwendung haben. Beispiele für bekannte MDI-Anwendungen sind Microsoft Excel oder der Programm-Manager und der Datei-Manager von Windows. Der MDI-Standard gehört auch zur bekannten IBM-Spezifikation *Common User Access* (CUA).

Die Komponenten einer MDI-Anwendung

Bestimmte Komponenten sind in jeder MDI-Anwendung vorhanden. Die wichtigste ist das Hauptfenster, das in diesem Zusammenhang Rahmenfenster heißt. Im Client-Bereich des Rahmenfensters steht ein nicht sichtbares Fenster, das MDI-Client-Fenster, das die untergeordneten Fenster enthält, die als untergeordnete MDI-Fenster bezeichnet werden. Dieses Fenster spielt eine wichtige Rolle, weil hier die Verwaltung der untergeordneten MDI-Fenster erfolgt.

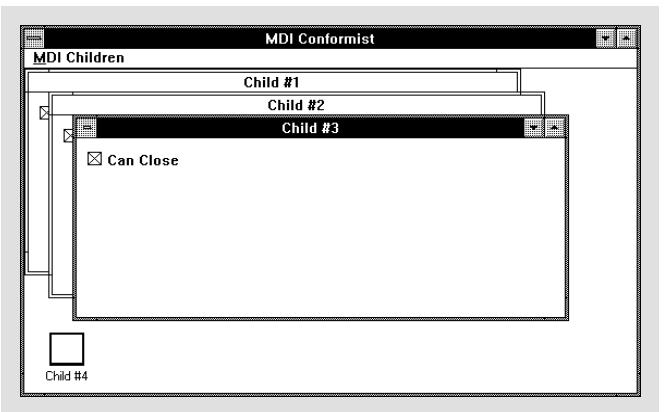


Abbildung 13.1 Die Komponenten einer MDI-Beispielanwendung

Das Rahmenfenster verfügt über ein Menü, das häufig den Titel Window trägt und das Einträge wie *Tile*, *Cascade*, *Arrange Icons* und *Close All* enthält. Jedes geöffnete untergeordnete MDI-Fenster wird an das Ende dieses Menüs angefügt. Das aktuell ausgewählte Fenster ist in diesem Menü mit einem Häkchen gekennzeichnet.

Jedes untergeordnete MDI-Fenster hat einige Eigenschaften von überlappenden Fenstern. Es kann auf die volle Größe des MDI-Client-Fensters vergrößert oder zu einem Symbol verkleinert werden, das oberhalb des unteren Rands des Rahmenfensters steht. Untergeordnete MDI-Fenster erscheinen niemals außerhalb ihres Rahmenfensters. Ein untergeordnetes MDI-Fenster kann kein Menü haben, so daß alle Funktionen über das Menü des Rahmenfensters gesteuert werden. Als Titel eines untergeordneten MDI-Fensters dient häufig der Name der mit ihm verknüpften geöffneten Datei. Diese Eigenschaft ist jedoch optional und steht unter Programmkontrolle. Sie können sich eine MDI-Anwendung als »Mini«-Windows-Sitzung mit vielen durch Fenster oder Symbole repräsentierten Anwendungen vorstellen.

Das MDI-Fenster - ein Objekt

ObjectWindows definiert Objekttypen für das MDI-Rahmenfenster und die Client-Fenster. Dabei handelt es sich um die Typen *TMDIWindow* und *TMDIClient*. *TMDIWindow* stammt von *TWindow* ab, während *TMDIClient* ein Steuerelement ist und von *TControl* abstammt.

In einer ObjectWindows-MDI-Anwendung ist das MDI-Client-Fenster seinem Rahmenfenster zugehörig und in seinem *ClientWnd*-Feld gespeichert. Das Rahmenfenster enthält alle seine untergeordneten Fenster in seiner verketteten Liste *ChildList*. Die untergeordneten MDI-Fenster sind Instanzen eines Objekttyps, der von dem von Ihnen geschriebenen *TWindow*-Typ abstammt.

Die *TMDIWindow*-Methoden befassen sich in erster Linie mit der Konstruktion und Verwaltung der untergeordneten MDI-Fenster und dem MDI-Client-Fenster sowie mit der Bearbeitung der Menüauswahlpunkte. Die wichtigste Aufgabe von *TMDIClient* ist die selbständige Verwaltung der untergeordneten MDI-Fenster.

Beim Entwurf von MDI-Anwendungen werden normalerweise neue Typen für Rahmen- und untergeordnete Fenster von *TMDIWindow* und *TWindow* abgeleitet.

Konstruieren von MDI-Fenstern

Bei der Konstruktion von Fenstern für eine MDI-Anwendung müssen Sie einige Besonderheiten beachten.

Eine vollständige Erläuterung der Konstruktion finden Sie in Kapitel 10 »Fensterobjekte«.

Konstruieren von MDI-Rahmenfenstern

Das Rahmenfenster einer MDI-Anwendung ist gleichzeitig das Hauptfenster der Anwendung. Es wird aus diesem Grund in der *InitMainWindow*-Methode seines Anwendungstyps konstruiert. Anders als bei *TWindow* hat der *Init*-Konstruktor von *TMDIWindow* kein übergeordnetes Fenster als Argument (Rahmenfenster sind Hauptfenster und haben kein übergeordnetes Fenster). Dafür hat er einen weiteren Parameter, nämlich ein Handle auf ein Menü.

Bei Hauptfenstern, die nicht zu einer MDI-Anwendung gehören und von *TWindow* abgeleitet sind, können Sie *Init* so definieren, daß in *Attr.Menu* ein gültiges Menü-Handle gespeichert wird. MDI-Rahmenfenster erfordern ein Menü, weshalb *TMDIWindow.Init* ein Menüargument benötigt, um *Attr.Menu* setzen zu können.

Das Menü des Rahmenfensters muß über ein dem MDI-Stil entsprechendes Menü für untergeordnete Fenster verfügen. Hier handelt es sich um das Menü, an dessen Ende die Titel der untergeordneten MDI-Fenster während der Anwendungsoperation angefügt werden. Dabei ist folgendes zu beachten: Das Rahmenfenster muß wissen, welcher Hauptmenüeintrag das Menü für die untergeordneten Fenster ist. *TMDIWindow*-Objekte speichern einen Integerwert als Positionsangabe im Objektfeld *ChildMenuPos*. *TMDIWindow.Init* initialisiert dieses Feld mit 0, womit der am weitesten links stehende Hauptmenüeintrag bezeichnet wird. Sie können *Init* jedoch für Ihre von *TMDIWindow* abgeleiteten Typen neu definieren, um *ChildMenuPos* neu zu setzen:

```
constructor MyMDIWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  TMDIWindow.Init(ATitle, AMenu);
  ChildMenuPos := 1;
end;
```

TMDIWindow.Init ruft auch *InitClientWindow* zur Konstruktion des *TMDIClient*-Objekts auf, das als MDI-Client-Fenster dient. *TMDIWindow.SetupWindow* erstellt das MDI-Client-Fenster.

Konstruieren von untergeordneten MDI-Fenstern

TMDIWindow definiert eine Antwortmethode mit dem Namen *CreateChild*, das über einen Menüpunkt aufgerufen wird, der die Befehl-ID *Create_Child* hat. So wie *CreateChild* von *TMDIWindow* definiert ist, konstruiert und erstellt es durch einen Aufruf von *TMDIWindow.InitChild* ein untergeordnetes MDI-Fenster vom Typ *TWindow*. Zur Festlegung des richtigen Typs für das untergeordnete Fenster (eines Nachkommen von *TWindow*) definieren Sie *InitChild* für den MDI-Rahmenfenstertyp neu:

```
function MyMDIWindow.InitChild: PWindowsObject;
begin
```

```
    InitChild := New(PMyChild, Init(@Self, 'New Child Window'));
  end;
```

Falls Ihr Rahmenfenster aber zu Beginn ein untergeordnetes MDI-Fenster anzeigen soll, muß für dieses die Größe explizit angegeben werden. Untergeordnete MDI-Fenster müssen in der *SetupWindow*-Methode des MDI-Rahmenfensters konstruiert und erstellt werden und nicht, wie andere untergeordnete Fenster, in dessen *Init*-Methode. In diesem Beispiel müssen Sie das untergeordnete Fenster auch explizit erstellen:

```
procedure MyMDIWindow.SetupWindow;
var
  ARect: TRect;
  NewChild: PMyChild;
begin
  TMDIWindow.SetupWindow;
  NewChild := PMyChild(InitChild);
  GetClientRect(HWindow, ARect);
  with NewChild^.Attr, ARect do
  begin
    W := (right * 4) div 5;
    H := (bottom * 3) div 5;
    Title := 'Child #1';
end;
  Application^.MakeWindow(NewChild);
end;
```

In manchen Anwendungen sollen untergeordnete MDI-Fenster als Reaktion auf die Auswahl nicht nur eines Menüpunkts erstellt werden. Beispielsweise zeigt ein Texteditor ein neues untergeordnetes Fenster mit dem Dateinamen als Titel an, wenn der Anwender *New* oder *Open* wählt. Definieren Sie für diesen Fall eine Botschaftsantwort-Methode, um das untergeordnete Fenster zu konstruieren.

Bearbeiten von Botschaften in MDI-Anwendungen

Wie bei regulären über- und untergeordneten Fenstern gelangen Windows-Botschaften, die auf Befehlen oder der ID des untergeordneten Fensters beruhen, zuerst zum untergeordneten Fenster, damit sie hier abgefangen und bearbeitet werden können. Danach wird die Botschaft an das übergeordnete Fenster weitergegeben. Bei MDI-Anwendungen gelangt die Botschaft jedoch zuerst an das aktive untergeordnete MDI-Fenster, dann zum MDI-Client-Fenster und schließlich zum MDI-Rahmenfenster (dem allen untergeordneten MDI-Fenstern übergeordneten Fenster). Auf diese Weise kann mit Hilfe des Rahmenfenstermenüs die Aktivität im aktuellen untergeordneten MDI-Fenster gesteuert werden. Danach haben Client- und Rahmenfenster die Möglichkeit, auf die Botschaft zu antworten.

Verwalten von untergeordneten MDI-Fenstern

Die MDI-Fenstertypen von ObjectWindows verfügen über Methoden für die Manipulation der untergeordneten MDI-Fenster in einer MDI-Anwendung. Obwohl ein sehr großer Teil der auszuführenden Aufgaben von *TMDIClient* ausgeführt wird, kann über die *TMDIWindow*-Methoden auf alle Funktionen und Daten zugegriffen werden.

Aktivierung von untergeordneten Fenstern

Der Anwender einer MDI-Anwendung kann jedes geöffnete oder minimierte untergeordnetes MDI-Fenster aktivieren. Sie wollen aber vielleicht eine Aktion ausführen, wenn der Anwender ein Fenster durch Aktivieren eines anderen deaktiviert. So könnten die Menüs des Rahmenfensters den aktuellen Status des aktiven untergeordneten Fensters durch graue Einfärbung oder Markierung anzeigen. Jedesmal, wenn ein untergeordnetes Fenster aktiviert oder deaktiviert wird, empfängt es die Windows-Botschaft *wm_MDIActivate*. Wenn Sie für diese Botschaft eine Botschaftsantwort-Methode für das untergeordnete Fenster definieren, können Sie immer verfolgen, welches untergeordnete Fenster aktiv ist und entsprechend antworten.

Das Menü für untergeordnete Fenster

TMDIWindow definiert Windows-Botschaftsantworten, die auf die Standardauswahlpunkte *Tile*, *Cascade*, *Arrange Icons* und *Close All* der MDI-Menüs antworten. Diese Methoden erwarten auf Befehlen basierende Botschaften mit vordefinierten Menü-ID-Konstanten. Stellen Sie sicher, daß diese IDs beim Aufbau einer Menüressource für das untergeordnete Fenster verwendet werden:

Tabelle 13.1 Standard-MDI-Aktionen, Befehle und Methoden

Aktion	Menü-ID-Konstante	TMDIWindow-Methode
Tile	*cm_TileChildren*	*CMTileChildren*
Cascade	*cm_CascadeChildren*	*CMCascadeChildren*
Arrange Icons	*cm_ArrangeChildIcons*	*CMArrangeChildIcons*
Close All	*cm_CloseChildren*	*CMCloseChildren*

Die Antwortmethoden von *TMDIWindow*, wie zum Beispiel *CMTileChildren*, rufen andere *TMDIWindow*-Methoden auf, wie zum Beispiel *TileChildren*. Diese Methoden rufen *TMDIClient*-Methoden desselben Namens auf, wie zum Beispiel *TMDIClient^.TileChildren*. Um Verhalten und Eigenschaften neu zu definieren, überschreiben Sie *TMDIWindow.TileChildren* bzw. andere *TMDIWindow*-Metho-

den. Bei untergeordneten MDI-Fenstern ist das Beantworten von auf Befehlen basierenden Botschaften, die vom Menü für untergeordnete Fenster erzeugt werden, kein geeignetes Vorgehen.

Beispiel für eine MDI-Anwendung

Das Programm *MDITest* produziert die MDI-Anwendung, die Sie in Abbildung 13.1 sehen.

Das vollständige Programm finden Sie unter dem Namen MDITEST.PAS auf Ihren Originaldisketten.

III. ObjectWindows für Fortgeschrittene

14. Speicher-verwaltung

Während der Programmierarbeit müssen Sie sich normalerweise nicht mit der Verwaltung des Speichers durch Windows befassen, da dies – zumindest größtenteils – von Turbo Pascal erledigt wird. Verwenden Sie einfach die Routinen *New* und *Dispose* (oder *GetMem* und *FreeMem*) zur Speicherbelegung für dynamische Variablen, der Compiler sorgt dann dafür, daß die erforderliche Reservierung vom globalen Heap von Windows erfolgt.

Sie können auch die ObjectWindows-Funktion *MemAlloc* verwenden, dann wird zusätzlich der Sicherheitsbereich überprüft. (*MemAlloc* und der Sicherheitsbereich werden in Kapitel 19 »Standardrichtlinien für Anwendungen« ausführlich besprochen.)

Trotzdem kann es in bestimmten Fällen nötig werden, sich direkt mit der Speicherverwaltung von Windows zu beschäftigen. Das vorliegende Kapitel geht auf diese Situationen ein und erläutert, wie der Speicher von Windows reserviert und verwendet wird.

Die Speicherverwaltung von Windows

Bei der Entwicklung von Turbo-Pascal-Programmen kann es Situationen geben, in denen der Speicherbedarf der Anwendung die Kapazität übersteigt, die vom compilierten Programm direkt zur Verfügung gestellt werden kann. Angenommen, Ihre Anwendung enthält eine Funktion, die eine Bitmap-Datei von der Festplatte lädt und sie auf dem Bildschirm angezeigt. Das Programm benötigt in diesem Fall einen Platz zum Speichern der Bitmap-Daten, bevor diese an Windows zur Anzeige übergeben werden. Eine Bitmap-Datei kann aber ziemlich groß sein – zu groß, um in einer Pascal-Variablen Platz zu finden. Zur Lösung dieses Problems verfügt Windows über Funktionen für die Belegung und Manipulation von Speicher, der außerhalb des Speicherbereichs des Programms liegt.

Windows ist ein multitasking-fähiges Programm, es können also mehrere Anwendungen gleichzeitig denselben Speicher benutzen. Der Speicher wird von Windows deshalb so verwaltet, daß er von jeder Anwendung immer optimal genutzt werden kann. Das vorliegende Kapitel geht ausführlich auf die verschiedenen Speicherarten von Windows ein und erläutert die Mechanismen, die die

Erstellung und den Zugriff auf Windows-Speicher innerhalb der Anwendung ermöglichen.

Die Verwaltung des Speichers durch Windows

Bei den meisten auf DOS basierenden Speicherverwaltungssystemen sind die belegten Speicherblöcke fest mit einer bestimmten Adresse verbunden. Dagegen können die Speicherblöcke in Windows auch *verschiebbar* und *verwerfbar* sein. Ein als verschiebbar reservierter Speicherblock hat keine feste Adresse und kann von Windows bei Bedarf verlagert werden. Zum Beispiel können zwei nichtzusammenhängende Blöcke des reservierten Speichers zusammengeführt werden, um einen größeren Block mit freiem Speicher zu erzeugen.

Verwerfbare Speicherblöcke können von Windows verschoben, aber auch durch die Neureservierung des Blocks auf die Länge 0 verworfen werden. Bei dieser Neureservierung werden die im Block enthaltenen Daten gelöscht. Die Anwendung muß deshalb den Speicherblock (falls gewünscht) neu reservieren und anschließend mit den korrekten Daten laden.

Das Verschieben oder Verwerfen von Speicherblöcken zu einer größeren, zusammenhängenden freien Speichereinheit wird als *Zusammenfassung* bezeichnet.

Abbildung 14.1 zeigt ein Beispiel für eine Zusammenfassung.

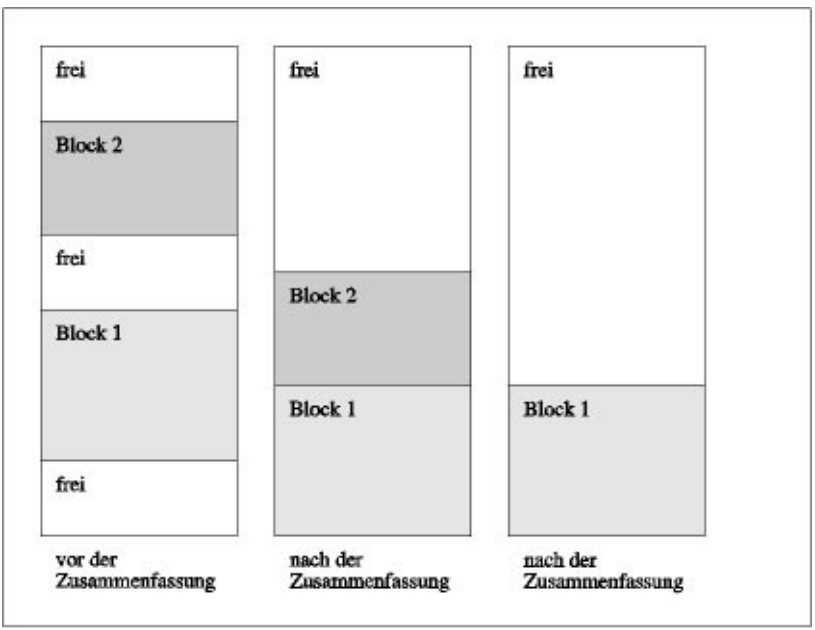


Abbildung 14.1 Beispiel für eine Zusammenfassung

Die linke Säule enthält drei nicht-zusammenhängende Blöcke freien Speichers, die durch die Blöcke 1 und 2 voneinander getrennt sind. Nehmen wir an, es wird Speicher angefordert, der größer als jeder einzelne der drei freien Speicherblöcke sein soll. Sind Block 1 und 2 verschiebbar, können die drei Blöcke zusammengefaßt werden. Daraus ergibt sich ein größerer zusammenhängender freier Speicherblock (mittlere Säule). Ist der Bedarf an freiem Speicher noch höher und ist Block 2 verwerfbar, kann er (durch Neureservierung mit der Länge 0) aus dem Speicher entfernt werden. Auf diese Weise ergibt sich ein noch größerer zusammenhängender Block mit freiem Speicher (rechte Säule).

Handles und Adressen

Möglicherweise glauben Sie jetzt, die Verwaltung des Windows-Speichers wäre sehr schwierig und aufwendig. Glücklicherweise ist dies nicht der Fall. Bei der Reservierung von Windows-Speicher erhält Ihre Anwendung keinen Zeiger, sondern ein Handle auf den betreffenden Speicherblock. Das Handle ist nicht die Adresse eines Speicherblocks, sondern ein eindeutiger Name, den die Anwendung in Funktionsaufrufen zur Speicherverwaltung als Referenz auf den Block benutzt. Sie können sich das Handle als Zeiger auf einen Zeiger vorstellen. Das Handle ist ungültig, wenn sein Wert 0 ist.

Ohne Adresse ist es aber unmöglich, auf die Daten in einem verschiebbaren Speicherblock zuzugreifen. Um eine Adresse zu erhalten, muß der Block gesperrt werden. Das Sperren eines Blocks liefert einen Zeiger auf den Anfang des Blocks. Der Zeiger behält seine Gültigkeit, solange der Block gesperrt ist. Dies gilt auch für verschiebbare und verwerfbare Blöcke. Nachdem der Zugriff auf die Daten erfolgt ist, sollte die Sperre vom Block entfernt werden. Wird dies vergessen, kann Windows den Block nicht mehr verschieben oder verwerfen, die Speicherverwaltung wird somit ineffizient. Nach dem Aufheben der Sperre ist der Zeiger des Blocks nicht mehr gültig. Ein ungültiger Zeiger ist nil.

Lokaler und globaler Speicher

Windows ermöglicht es Ihrer Anwendung, Speicherblöcke zu erzeugen und zu benutzen. Der Speicher dafür kann aus zwei Quellen kommen: dem lokalen Heap und dem globalen Heap. Beim globalem Heap handelt es sich um den Speicher, der frei (nicht belegt) ist und von allen Anwendungen genutzt werden kann. Windows, Windows-Anwendungen und von Windows-Anwendungen belegte globale Speicherblöcke befinden sich alle im globalen Heap. Der Speicher im lokalen Heap kann nur von Ihrer Anwendung genutzt werden.

Bei jeder Speicherreservierung muß angegeben werden, ob lokaler oder globaler Speicher genutzt werden soll. Für die Wahl sind mehrere Faktoren ausschlaggebend. Die Vor- und Nachteile der verschiedenen Speicherarten werden im folgenden kurz aufgelistet:

Lokaler Speicher

- Schneller Zugriff.
- Begrenzte Größe (unter 64 KByte).

Globaler Speicher

- Blöcke sind an 32-Byte-Grenzen ausgerichtet.
- Windows reserviert zwanzig Bytes pro Block für den Overhead.
- Systemweite Begrenzung für Handles im Standard- und im erweiterten 386-Modus.
- Langsamer Zugriff.
- Mehr Speicher verfügbar.

Der entscheidende Faktor für Ihre Wahl ist normalerweise die Größe des Speichers, der zur Ausführung der betreffenden Aufgabe benötigt wird. Ist mehr als 1 KByte erforderlich, belegen Sie Speicher vom globalen Heap. Werden dagegen kleinere Blöcke für einen kurzen Zeitraum benötigt, sollte der Speicher vom lokalen Heap stammen.

Der lokale Heap

Auf den Speicher des lokalen Heap kann nur eine bestimmte Instanz Ihrer Anwendung zugreifen. Normalerweise wird in Windows der lokale Heap im Datensegment der Anwendung eingerichtet. Das Datensegment enthält außerdem den Stack, globale Programmvariablen und globale Variablen aller vom Programm genutzten Units. Der übrige Speicher im Datensegment (bis zu 64 KByte) kann als lokaler Heap genutzt werden.

Windows richtet den lokalen Heap für eine Anwendung nicht automatisch ein. Sie müssen deshalb die Größe des lokalen Heap während der Compilierung mit dem zweiten Parameter der Compiler-Direktive *$M* festlegen (voreingestellt sind 8 KByte).

Ein ausführliche Beschreibung dieser Direktive finden Sie in Kapitel 21 »Compiler-Direktiven« des *Programmierhandbuches*.

Die maximale Größe, die für den lokalen Heap angegeben werden kann, hängt auch davon ab, ob das Datensegment fest oder verschiebbar ist.

- Ist es fest, kann nur ein lokaler Block belegt werden, der die Größe des Heap, die beim Compilieren und Linken angegeben wurde, nicht überschreitet.
- Ist das Datensegment verschiebbar und wird mehr Speicher angefordert, als für den lokalen Heap festgelegt wurde, versucht Windows, diesen Bedarf durch die Reservierung zusätzlichen Speichers (bis zur 64 KByte-Grenze) zu erfüllen. Windows kann dazu das Datensegment verschieben, wobei allerdings alle Zeiger auf lokale Daten ungültig werden.

Reservierung und Verwendung von lokalem Speicher

Die Windows-Funktion *LocalAlloc* reserviert einen Speicherblock von der im Funktionsaufruf festgelegten Art und Größe. Es muß angegeben werden, ob der Block fest, verschiebbar oder verschiebbar und verwerfbar sein soll (ein verwerfbarer Block muß gleichzeitig auch verschiebbar sein). Dafür stehen folgende Optionen zur Verfügung, die im *Flags*-Parameter des Funktionsaufrufs angegeben werden müssen:

lmem_Fixed erzeugt einen festen Speicherblock.

lmem_Moveable erzeugt einen verschiebbaren Speicherblock.

lmem_Moveable or *lmem_Discardable* erzeugt einen verwerfbaren Speicherblock.

Wird im lokalen Heap ein neuer Block belegt, können dadurch andere Blöcke verschoben oder verworfen werden. Um das Verwerfen zu verhindern, können Sie zum *Flags*-Parameter *lmem_NoDiscard* hinzufügen. Mit *lmem_NoCompact* können Sie verhindern, daß Blöcke verschoben oder verworfen werden.

War die Reservierung erfolgreich, liefert die Funktion *LocalAlloc* ein *Handle* auf den Speicherblock. Andernfalls ist das Ergebnis 0. Bei einer erfolgreichen Reservierung hat der Speicherblock mindestens die geforderte Größe, er kann aber auch größer sein. Mit der Funktion *LocalSize* kann seine tatsächliche Größe ermittelt werden. Die maximale Größe für einen lokalen Block ist von der Größe des lokalen Heap abhängig.

Sie verfügen nun zwar über ein Handle auf den Block, können aber trotzdem noch nicht auf seine Daten zugreifen. Der Block muß zuerst gesperrt werden, um seine Adresse zu erhalten. Dies geschieht durch einem Aufruf der Funktion *LocalLock*. *LocalLock* fixiert den Block zeitweise an einer bestimmten Adresse im lokalen Heap und liefert einen Zeiger auf den Block.

Diese Adresse bleibt solange gültig, bis die Sperre des Blocks mit der Funktion *LocalUnlock* aufgehoben wird. *LocalLock* inkrementiert den Sperrenzähler des Blocks. Der Sperrenzähler gibt an, wie oft der betreffende Block bereits fixiert wurde.

Der von der Funktion *LocalLock* zurückgelieferte Wert muß nicht unbedingt ein gültiger Zeiger sein – Sie sollten ihn deshalb überprüfen.

Der Rückgabewert ist nil, wenn das Handle, das ihr übergeben wurde, ungültig ist oder der Block verworfen wurde (dazu muß für den Block das Attribut *lmem_Discardable* angegeben worden sein).

Das folgende Programm reserviert einen Block von 256 Bytes im lokalen Speicher.

```
var
  LocalHandle: THandle;
  PtrLocalData: Pointer;

begin
  LocalHandle := LocalAlloc(lmem_Moveable or lmem_Discardable, 256);
  if LocalHandle <> 0 then
```

```
  begin
    PtrLocalData := LocalLock(LocalHandle);
    if PtrLocalData <> nil then
    begin
      { Bearbeiten Sie hier die Daten mit Hilfe des Zeigers }
      LocalUnlock(LocalHandle);
    end
    else { Block wurde nicht gesperrt };
  end
  else { Block wurde nicht reserviert };
end;
```

Wird ein lokaler Block mit dem Attribut *lmem_Fixed* reserviert, kann er nicht verschoben werden. Es ist deshalb nicht erforderlich, *LocalLock* zum Sperren eines Objekts an einer festen Adresse aufzurufen. Nur in diesem Fall ist das von *LocalAlloc* zurückgelieferte 16-Bit-Handle auch die 16-Bit-Adresse des Speicherblocks.

Freigeben und verwerfen lokaler Speicherblöcke

Lokale Speicherblöcke können mit der Funktion *LocalFree* freigegeben werden. Durch die Freigabe wird der Inhalt des Blocks vom lokalen Heap und sein Handle aus der Tabelle mit gültigen Speicher-Handles entfernt. Dieses Handle kann dann nicht erneut verwendet werden. Speicherblöcke, die nicht mehr benötigt werden, sollten freigegeben werden. Außerdem sollte bei der Beendigung einer Anwendung die Freigabe aller Speicherblöcke erfolgen.

Mit der Funktion *LocalDiscard* kann ein lokaler Speicherblock explizit verworfen werden. Beim Verwerfen eines lokalen Blocks wird er auf die Größe 0 reduziert und sein Inhalt vom lokalen Heap entfernt. Sein Handle bleibt jedoch gültig und kann durch einen Aufruf der Funktion *LocalReAlloc* mit dem Handle und einer neuen Größe erneut verwendet werden. Sie können die Verarbeitungszeit etwas verkürzen, wenn Sie Handles verwerfen und wieder reservieren anstatt sie freizugeben und neu zu reservieren.

```
var
  LocalHandle: THandle;
begin
  { LocalHandle wurde bereits reserviert }
  LocalDiscard(LocalHandle);
  { jetzt wird das Handle wieder benutzt,
    um einen neuen Speicherblock zu belegen }
  LocalHandle := LocalReAlloc(LocalHandle, 256, lmem_Moveable);
  ...
end;
```

Ein lokaler Speicherblock kann nur freigegeben oder verworfen werden, wenn sein Sperrenzähler 0 ist.

Neureservieren und Ändern lokaler Speicherblöcke

Mit der Funktion *LocalReAlloc* ist es möglich, die Größe eines Blocks zur ändern und gleichzeitig seinen Inhalt beizubehalten. Windows kürzt den Block, wenn eine geringere Größe angegeben wird. Wird dagegen die Größe erhöht und *lmem_ZeroInit* angegeben, wird der zusätzliche Bereich mit Nullen initialisiert. Bei einer Vergrößerung ohne Angabe von *lmem_ZeroInit* enthält der Block undefinierte Daten. Wie bei *LocalAlloc* können Blöcke auch bei einer Neureservierung mit *LocalReAlloc* verschoben oder verworfen werden. Durch die Verwendung von *lmem_NoDiscard* oder *lmem_NoCompact* kann das Verwerfen oder Verschieben anderer Blöcke während der Neureservierung verhindert werden.

Mit *LocalReAlloc* können auch die Attribute des Blocks von *lmem_Moveable* zu *lmem_Discardable* (und umgekehrt) geändert werden. Dazu geben Sie zusätzlich zum neuen Attribut *lmem_Modify* an. *LocalReAlloc* kann aber nicht zusammen mit *lmem_Modify* verwendet werden, um das Attribut *lmem_Fixed* hinzuzufügen oder zu entfernen. Das folgende Programmfragment macht einen verschiebbaren Block verwerfbar:

```
var
  LocalHandle: THandle;

begin
  LocalHandle := LocalAlloc(lmem_Moveable, 256);
  LocalHandle := LocalReAlloc(localHandle, 256, lmem_Modify or
    lmem_Discardable);
end;
```

Abfragen lokaler Speicherblöcke

Mit den Funktionen *LocalSize* und *LocalFlags* können Informationen über lokale Speicherblöcke abgerufen werden. *LocalSize* liefert die Größe des lokalen Blocks zurück. Bei der Speicherreservierung wird möglicherweise ein Block erzeugt werden, der die angeforderte Größe überschreitet. Mit *LocalSize* läßt sich seine tatsächliche Größe wie folgt feststellen:

```
var
  LocalHandle: THandle;
  BlockSize: Word;

begin
  LocalHandle := LocalAlloc(lmem_Moveable, 256);
  if LocalHandle <> 0 then BlockSize := LocalSize(LocalHandle);
end;
```

Mit *LocalFlags* läßt sich der Sperrenzähler des Speicherblocks überprüfen. Außerdem kann mit dieser Funktion ermittelt werden, ob ein Block verwerfbar ist und ob er verworfen wurde.

```
var
  LocalHandle: THandle;
  Flags, LockCount: Word;

begin
  { Der Block muß bereits reserviert und kann auch gesperrt sein }
  Flags := LocalFlags(LocalHandle);
  LockCount := Flags and lmem_LockCount;
end;
```

Mit der Funktion *LocalCompact* kann die Größe des größten freien Speicherblocks im lokalen Heap ermittelt werden. Dazu wird als Parameter 0 angegeben.

Hinweise für die Programmierung

Verschiebbare und verwerfbare Speicherblöcke sollten möglichst sofort nach ihrer Benutzung entsperrt werden. Dies ist besonders dann wichtig, wenn weitere Blöcke im lokalen Heap reserviert werden sollen. Wird die Sperre für diese Blöcke nicht aufgehoben, kann Windows sie nicht verschieben, um für erneute Reservierungen Platz zu schaffen. Vor dem Programmende sollten alle Speicherblöcke freigegeben werden.

Der globale Heap

Als globaler Heap wird der systemweite Speicher bezeichnet, der von Windows und Ihren Programmen genutzt werden kann. Wieviel Speicher des globalen Heap für Ihre Anwendung reserviert werden kann, hängt von dem Modus ab, in dem Windows läuft. Da der globale Speicher von mehreren Programmen gemeinsam genutzt wird, muß Ihre Anwendung so effizient wie möglich damit umgehen. Andernfalls könnte die Ausführungsgeschwindigkeit im gesamten System darunter leiden.

Reservierung und Verwendung von globalem Speicher

Die Funktion *GlobalAlloc* reserviert einen Speicherblock von einer bestimmten Art und Größe, die mit den Argumenten des Funktionsaufrufs festgelegt werden müssen. Wie bei der Zuweisung von lokalem Speicher muß angegeben werden, ob der Block fest, verschiebbar oder verschiebbar und verwerfbar sein soll (ein verwerfbarer Block muß gleichzeitig auch verschiebbar sein). Dafür stehen folgende Optionen zur Verfügung, die im *Flags*-Parameter des Funktionsaufrufs angegeben werden müssen:

gmem_Fixed erzeugt einen festen Speicherblock.

gmem_Moveable erzeugt einen verschiebbaren Speicherblock.

gmem_Moveable or *gmem_Discardable* erzeugt einen verwerfbaren Speicherblock.

Bei der Reservierung eines neuen Blocks im globalen Heap können andere Blöcke verschoben oder verworfen werden. Um das Verwerfen zu verhindern, können Sie zum Flags-Parameter gmem_NoDiscard hinzufügen. Mit gmem_NoCompact können Sie verhindern, daß Blöcke verschoben oder verworfen werden.

War die Reservierung erfolgreich, liefert die Funktion *GlobalAlloc* ein *Handle* auf den Speicherblock. Andernfalls ist das Ergebnis 0. Der von *GlobalAlloc* zurückgelieferte Wert sollte immer überprüft werden, um festzustellen, ob die Reservierung erfolgreich war. Bei einer erfolgreichen Reservierung hat der Speicherblock mindestens die geforderte Größe, möglicherweise ist er größer. Mit der Funktion *GlobalSize* können Sie den Block auf sein Größe überprüfen. Die maximale Größe für einen globalen Block liegt im Standardmodus bei 1 MByte, im erweiterten 386-Modus bei 64 MByte.

Um die Adresse eines globalen Speicherblocks zu erhalten, muß er gesperrt werden. Dies geschieht durch einen Aufruf der Funktion *GlobalLock*. Im Real Mode fixiert *GlobalLock* den Block zeitweise an einer bestimmten Adresse im globalen Heap. In den anderen Modi wird er nicht fixiert. *GlobalLock* liefert in allen Modi einen far-Zeiger auf den Block. Dieser bleibt solange gültig, bis die Sperre des Blocks mit der Funktion *GlobalUnlock* aufgehoben wird.

Der von *GlobalLock* zurückgelieferte Wert muß nicht unbedingt ein gültiger Zeiger sein – Sie sollten ihn deshalb überprüfen. Der Rückgabewert ist nil, wenn das Handle, das dieser Funktion übergeben wurde, ungültig ist oder der Block verworfen wurde (dazu mußte für den Block das Attribut *gmem_Discardable* angegeben worden sein). Das folgende Programm reserviert einen Block im globalen Speicher.

```
type
  PBigArray = ^TBigArray;
  TBigArray = array[0..20000] of Byte;
var
  GlobalHandle: THandle;
  PtrGlobalData: PBigArray;
  Value: Byte;

begin
  GlobalHandle := GlobalAlloc(gmem_Moveable, 20000);
  if GlobalHandle <> 0 then
  begin
    PtrGlobalData := GlobalLock(GlobalHandle);
    if PtrGlobalData <> nil then
    begin
      { Bearbeiten Sie hier die Daten mit Hilfe des
        Zeigers PtrGlobalData }
      GlobalUnlock(GlobalHandle);
    end
```

```
      else { Block wurde nicht gesperrt };
    end
    else { Block wurde nicht reserviert };
  end;
```

Im Real Mode muß *GlobalLock* den Block im Speicher fixieren, bis die Sperre aufgehoben wird. Um anzuzeigen, daß der Block tatsächlich fixiert ist und nicht freigegeben oder verworfen werden kann, wird der Sperrenzähler um 1 erhöht. Der Aufruf von *GlobalUnlock* dekrementiert den Sperrenzähler um 1. Im Standardmodus und im erweiterten 386-Modus wird der Sperrenzähler nicht erhöht, da der Block nicht tatsächlich im Speicher fixiert ist (dies gilt nicht für verwerfbare Blocks). Der Zeiger ist immer gültig, auch dann, wenn Windows den Block im Speicher verschiebt.

Im Gegensatz zu lokalen Speicherblöcken muß ein Block, der mit dem *gmem_Fixed*-Attribut reserviert wurde, vor einem Zugriff auf seine Daten fixiert werden, da in diesem Fall das von *GlobalAlloc* gelieferte Handle kein Zeiger auf den Block ist.

Weitere Möglichkeiten zum Sperren eines globalen Blocks

Zum Sperren und zum Aufheben der Sperre von globalen Speicherblöcken stehen weitere Funktionen zur Verfügung. *GlobalFix* sperrt den Block im Speicher und verhindert sein Verschieben. Dabei ist es gleichgültig, in welchem Modus sich Windows gerade befindet. Da der Block in diesem Fall fixiert ist, wird der Sperrenzähler um 1 erhöht.

Die Verwendung von *GlobalFix* kann die Zusammenfassung des Speichers durch Windows beeinträchtigen. Setzen Sie diese Funktion deshalb nur dann ein, wenn es unbedingt erforderlich ist. Das Fixieren von Speicherblöcken mit *GlobalFix* ist nur selten nötig, da bei globalen Blöcken der Zeiger auch nach dem Verschieben noch gültig ist.

GlobalUnfix entsperrt den Block und dekrementiert seinen Sperrenzähler.

GlobalWire verlagert ein Segment in den unteren Speicherbereich und sperrt es. Der Sperrenzähler wird dabei um 1 erhöht.

Diese Funktion eignet sich besonders für Speicherblöcke, die für einen längere Zeitraum gesperrt bleiben sollen. Da *GlobalWire* einen Block im unteren Speicherbereich plaziert, wird die Zusammenfassung des Speichers nicht beeinträchtigt. *GlobalUnWire* entfernt die Sperre von dem Block und dekrementiert den Sperrenzähler.

Freigeben und verwerfen von globalen Speicherblöcken

Globale Speicherblöcke werden auf die gleiche Weise freigegeben und verworfen wie lokale Blöcke. Der einzige Unterschied liegt in den Funktionsnamen: *GlobalFree* und *GlobalDiscard*.

Neureservieren und Ändern globaler Speicherblöcke

Mit der Funktion *GlobalReAlloc* kann die Größe eines Blocks geändert und gleichzeitig sein Inhalt beibehalten werden. Windows kürzt den Block, wenn eine geringere Größe angegeben wird. Wird dagegen die Größe erhöht und *gmem_ZeroInit* angegeben, wird der zusätzliche Bereich mit Nullen initialisiert. Bei einer Vergrößerung ohne Angabe von *gmem_ZeroInit* enthält der Block undefinierte Daten. Wie bei *GlobalAlloc* können Blöcke auch bei einer Neureservierung mit *GlobalReAlloc* verschoben und verworfen werden. Durch die Verwendung von *gmem_NoDiscard* oder *gmem_NoCompact* kann das Verwerfen oder Verschieben von Blöcken während der Neureservierung verhindert werden.

Mit *GlobalReAlloc* können auch die Attribute eines Blocks von *gmem_Moveable* zu *gmem_Discardable* (und umgekehrt) geändert werden. Dazu geben Sie neben dem neuen Attribut *gmem_Modify* an. *GlobalReAlloc* kann auch zusammen mit *gmem_Modify* verwendet werden, um das Attribut *gmem_Fixed* hinzuzufügen oder zu entfernen (bei lokalen Blöcken ist dies nicht möglich).

Wenn Sie für einen Speicherbereich eine neue Größe reservieren, so daß ein Vielfaches einer 64-KByte-Grenze überschritten wird, kann Windows ein neues Handle für diesen Block zurückliefern. Wenn Ihre Anwendung im Standardmodus läuft, trifft das auf die Neureservierung eines Blocks zu, die ein Vielfaches von 65519 Bytes überschreitet. Das alte Handle muß deshalb beibehalten und das von *GlobalReAlloc* zurückgelieferte neue Handle überprüft werden. Ist der Aufruf von *GlobalReAlloc* erfolgreich, müssen Sie das neue Handle dem alten zuweisen, andernfalls müssen Sie das alte Handle verwenden.

```
var
  GlobalHandle: THandle;
  TempHandle: THandle;

begin
  { Für TempHandle wurden ursprünglich 32K reserviert }
  { Durch Neureservierung wird eine 64K-Grenze überschritten }
  TempHandle := GlobalReAlloc(GlobalHandle, 102400,
    gmem_Moveable);
  if TempHandle <> 0 then
    GlobalHandle := TempHandle; { Es könnte sich um ein anderes
                                  Handle handeln }
  else { Block wurde nicht neu reserviert }
end;
```

Abfragen globaler Speicherblöcke

Mit den Funktionen *GlobalSize* und *GlobalFlags* können Informationen über globale Speicherblöcke abgerufen werden. *GlobalSize* liefert die Größe des glo-

balen Blocks zurück. Mit *GlobalFlags* läßt sich der Sperrenzähler des Speicherblocks überprüfen. Außerdem kann mit dieser Funktion ermittelt werden, ob ein Block verwerfbar ist und ob er verworfen wurde. *GlobalFlags* zeigt auch an, ob ein Block als *gmem_DDEShare* oder als *gmem_Not_Banked* reserviert wurde.

```
var
  globalHandle: THandle;
  flags: Word;

begin
  Flags := GlobalFlags(globalHandle);
  { Ist der Block verwerfbar? }
  if Flags and gmem_Discardable then { ja }
end;
```

Mit der Funktion *GlobalCompact* kann die Größe des größten freien Speicherblocks im globalen Heap ermittelt werden. Dazu wird als Parameter 0 angegeben. Es findet in diesem Fall keine Zusammenfassung statt. Der zurückgelieferte Wert gibt die Größe des größten freien Blocks an, der nach einer Zusammenfassung entstehen würde. Mit *GlobalCompact* kann auch das Verwerfen aller nicht gesperrter verwerfbarer Blöcke im System erzwungen werden. Geben Sie dazu -1 als Parameter an.

```
var
  globalHandle: THandle;
  largestSize: Longint;
begin
  largestSize := GlobalCompact(Long(0));
  GlobalCompact(Long(-1);  { Erzwingt das Verwerfen aller
                             ungesperrten, verwerfbaren Blöcke }
end;
```

Ändern des globalen Verwerfens

Windows stellt mit Hilfe eines LRU-Algorithmus (least-recently-used) fest, welcher verwerfbare Block im Bedarfsfall zuerst verworfen wird. Mit den Funktionen *GobalLRUOldest* und *GobalLRUNewest* können Sie die Position eines Speicherblocks in der Liste mit den zu verwerfenden Blöcken angeben. *GobalLRUOldest* teilt Windows mit, daß der als Parameter übergebene Block als nächstes verworfen werden soll. *GobalLRUNewest* plaziert den angegebenen Block an das Ende der Liste mit den zu verwerfenden Blöcken. Er wird demnach als letzter Block verworfen.

Speicherknappheit

Wenn Windows feststellt, daß es zu viel Zeit für das Zusammenfassen von Speicherblöcken verwendet, sendet es eine *wm_Compacting*-Botschaft an die

Hauptfenster der momentan laufenden Anwendungen. Diese Botschaft teilt den Anwendungen mit, daß die Speicherkapazität des Systems bald erschöpft ist, und weist sie an, sofort so viel globalen Speicher wie möglich freizugeben.

Hinweise für die Programmierung

Globale Speicherblöcke sollten möglichst sofort nach ihrer Verwendung entsperrt werden. Wenn die Sperre für diese Blöcke nicht aufgehoben wird, kann Windows sie nicht verschieben, um für erneute Reservierungen Platz zu schaffen. Vor dem Verlassen des Programms sollten alle Speicherblöcke freigegeben werden.

Aufgrund des Overhead und der Byte-Ausrichtung im globalen Speicher sollten Sie die Reservierung von kleinen globalen Speicherblöcken (unter 128 Bytes) vermeiden. Die Reservierung einer großen Anzahl von globalen Blöcken ist ebenfalls nicht zu empfehlen. Verwenden Sie stattdessen weniger, aber dafür größere Blöcke.

Übergeben Sie nie normale Handles für globale Speicherblöcke an andere Anwendungen, um gemeinsam genutzten Speicher bereitzustellen. Verwenden Sie zu diesem Zweck den dynamischen Datenaustausch und die Zwischenablage. Näheres finden Sie in Kapitel 16 »Dynamischer Datenaustausch«.

15. Dynamische Linkbibliotheken

Dynamische Linkbibliotheken (DLLs) sind ausführbare Module mit Code und/oder Daten, die zur Laufzeit zu einer Anwendung gelinkt werden können. Mit Hilfe von DLLs kann der Entwickler einzelne Komponenten einer Anwendung aufbauen und verwalten, indem er die Anwendungen um Fähigkeiten erweitert, ohne die compilierte Anwendung ändern zu müssen. DLLs bieten »gebrauchsfertige« Funktionseinheiten an, wie beispielsweise Dateiformatfilter, die in eine Anwendung eingebracht werden können. Diese Bibliotheken nützen Speicherplatz effizient aus, da sie von vielen Anwendungen gleichzeitig gemeinsam genutzt werden können.

Einzelheiten über die Struktur und den Gebrauch von DLLs finden Sie in Kapitel 10 »Dynamic-link libraries« des *Programmierhandbuches*. Im vorliegenden Kapitel befassen wir uns speziell mit dem Einsatz von Linkbibliotheken in Windows-Anwendungen.

Zugriff auf DLL-Routinen

Eine Linkbibliothk besteht aus einer Bibliothek mit Funktionen. Sie kann in den verschiedensten Programmiersprachen geschrieben sein und ist unabhängig davon, mit welcher Sprache auf sie zugegriffen wird. Die Deklaration einer Linkbibliotheksfunktion in einem Turbo-Pascal-Programm ähnelt der Deklaration einer externen Routine.

```
function MyExternalRoutine(Paramter1: Word): Word; far;
external 'MYDLL' index 12;
```

Die obige Funktionsdefinition deklariert *MyExternalRoutine* als Funktion in einem externen Linkbibliotheksmodul mit dem Ordinalwert 12. Ist das angegebene Modul nicht bereits geladen, wenn die dieses Modul deklarierende Anwendung geladen wird, lädt Windows das angegebene Modul in den Speicher.

Da während der Laufzeit gelinkt wird, definiert der Ordinalwert einen Offset-Wert in das Modul, über den die Routine gefunden wird. Die meisten der auf dem Markt befindlichen DLLs dokumentieren die Ordinalwerte ihrer Routinen.

Ein einfaches Beispiel für eine DLL

Im folgenden finden Sie ein Beispiel für eine DLL, die einige einfache mathematische Berechnungen ausführt. Sie setzt sich aus drei Dateien zusammen:

MATHDLL.PAS, eine Bibliothek, die die Berechnungsroutinen enthält.

MATH.PAS, eine Unit, die die Schnittstelle zu *MathDLL* definiert.

DLLTEST.PAS, ein Hauptprogramm, das die Unit *Math* verwendet.

Im folgenden sehen Sie MATHDLL.PAS

```
library MathDll;

uses WinTypes, WinProcs;

function Power(X, Y: Real): Real; export;
begin
  Power:=Exp(Y * ln(X));
end

function Payments(Period, Interest, Term, Principal: Real):
                  Real; export;
begin
  Payments := (Principal * Interest / Period) /
    (1 - Power(1 + Interest / Period, -Term * Period));
end;

function Principals(Payment, Period, Interest, Term: Real):
                    Real; export;
begin
  Principals := Payment * ((1 - Power(1 + Interest / Period,
    -Term * Period)) / (Interest / Period));
end;

procedure WriteError(Window: hWnd; ErrorMessage: PChar); export;
var
  S: String;
begin
  MessageBox(Window, ErrorMessage, 'Error', MB_OK);
end;

exports
  Power        index 1,
  Payments     index 2,
  Principals   index 3,
  WriteError   index 4;

begin
end.
```

Wenn Sie diese Bibliotheksroutinen in einem Turbo-Pascal-Programm verwenden wollen, müssen Sie eine Unit erzeugen, die die Routinen aus der DLL importiert. Im folgenden sehen Sie eine Unit, die alle vier in *MathDLL* vorhandenen Routinen verfügbar macht (es handelt sich um die Datei MATH.PAS):

```
unit Math;

interface

function Power(X, Y: Real): Real;
function Payments(Period, Interest, Term, Principal: Real): Real;
function Principals(Payment, Period, Interest, Term: Real): Real;
procedure WriteError(Window: Word; ErrorMessage: PChar);

implementation

function Power;         external 'MathDll' index 1;
function Payments;      external 'MathDll' index 2;
function Principals;    external 'MathDll' index 3;
procedure WriteError;   external 'MathDll' index 4;

end.
```

Beachten Sie, daß alle Routinen implizit vom Typ far sind, da sie im interface-Abschnitt der Unit aufgeführt sind.

Damit Ihre Anwendung die Bibliotheksfunktionen verwenden kann, muß lediglich *Math* im uses-Abschnitt angegeben werden. Ihre Anwendung muß nicht wissen, daß die aufzurufenden Routinen tatsächlich in einer Bibliothek stehen.

Das folgende Programm (DLLTEST.PAS) verwendet die Unit *Math* zur Ausführung von Berechungen:

```
program DLLTest;

{$R MATH}

uses WinTypes, WinProcs, WObjects, Math;

type
  TMyApplication = Object(TApplication)
    procedure InitMainWindow; Virtual;
  end;

  TPaymentDlg = Object(TDialog)
    Period, Interest, Term, Principal : Real;
    function LoadFields: Boolean;
    procedure OK(var Msg: TMessage); virtual id_First + IdOk;
  end;

  TPrincipalDlg = Object(TDialog)
    Payment, Period, Interest, Term: Real;
    function LoadFields: Boolean;
    procedure OK(var Msg: TMessage); virtual id_First + IdOk;
  end;
```

```
  PMyWindow = ^TMyWindow;
  TMyWindow = Object(TWindow)
    PaymentDlg: TPaymentDlg;
    PrincipalDlg: TPrincipalDlg;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure Payment(var Msg: TMessage); virtual cm_First + 201;
    procedure Principal(var Msg: TMessage); virtual cm_First + 202;
  end;

function TPaymentDlg.LoadFields: Boolean;
var
  E: Integer;
  S: String;
begin
  LoadFields:=false;
  S[0]:=Char(GetDlgItemText(HWindow, 104, @S[1], 20));
  Val(S, Interest, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 105, @S[1], 20));
  Val(S, Term, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 112, @S[1], 20));
  Val(S, Period, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 106, @S[1], 20));
  Val(S, Principal, E);
  If E<>0 then
    Exit;
  LoadFields:=true;
end;

procedure TPaymentDlg.Ok(var Msg: TMessage);
var
  S: String;
begin
  if not LoadFields then
    WriteError(HWindow, 'All fields must have values')
  else
    begin
      Str(Payments(Period, Interest, Term, Principal):10:2, S);
      S:=S+#0;
      While S[1]=' ' do Delete(S, 1, 1);
      SetDlgItemText(HWindow, 113, @S[1]);
    end;
end;
```

```
function TPrincipalDlg.LoadFields: Boolean;
var
  E: Integer;
  S: String;
begin
  LoadFields:=false;
  S[0]:=Char(GetDlgItemText(HWindow, 104, @S[1], 20));
  Val(S, Interest, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 105, @S[1], 20));
  Val(S, Term, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 112, @S[1], 20));
  Val(S, Period, E);
  If E<>0 then
    Exit;
  S[0]:=Char(GetDlgItemText(HWindow, 106, @S[1], 20));
  Val(S, Payment, E);
  If E<>0 then
    Exit;
  LoadFields:=true;
end;

procedure TPrincipalDlg.Ok(var Msg: TMessage);
var
  S: String;
begin
  if not LoadFields then
    WriteError(HWindow, 'All fields must have values')
  else
    begin
      Str(Principals(Payment, Period, Interest, Term):10:2, S);
      S:=S+#0;
      While S[1]=' ' do Delete(S, 1, 1);
      SetDlgItemText(HWindow, 113, @S[1]);
    end;
end;

constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'Menu');
end;

procedure TMyWindow.Payment(var Msg: TMessage);
begin
```

```
  PaymentDlg.Init(@Self, 'PaymentDlg');
  PaymentDlg.Execute;
  PaymentDlg.Done;
end;

procedure TMyWindow.Principal(var Msg: TMessage);
begin
  PrincipalDlg.Init(@Self, 'PrincipalDlg');
  PrincipalDlg.Execute;
  PrincipalDlg.Done;
end;

procedure TMyApplication.InitMainWindow;
begin
  MainWindow:=New(PMyWindow, Init(nil, 'Title'));
end;

var
  MyApplication: TMyApplication;
begin
  MyApplication.Init('DLLTest');
  MyApplication.Run;
  MyApplication.Done;
end.
```

16. Dynamischer Datenaustausch

Da unter Windows mehrere Programme gleichzeitig ausgeführt werden können, ist es von Vorteil, wenn diese Programme zur Laufzeit Daten gemeinsam nutzen. Windows unterstützt den Datentransfer zwischen den Anwendungen und stellt dazu eine Zwischenablage, ein Protokoll für den dynamischen Datenaustausch (DDE) und die Möglichkeit zur Definition neuer Windows-Botschaften zur Verfügung.

Die Zwischenablage von Windows

In der Zwischenablage können Daten in verschiedenen Formaten gespeichert werden. Jede Windows-Anwendung, die Daten in die Zwischenablage plaziert, kann dazu ein vordefiniertes Format wählen oder ein eigenes Format definieren. Lädt eine Anwendung Daten aus der Zwischenablage, kann sie durch eine Abfrage das Format der gespeicherten Daten feststellen. Dieselben Daten können in unterschiedlichen Formaten in der Zwischenablage gespeichert werden, um der aufrufenden Anwendung die Wahl zwischen verschiedenen Formaten zu ermöglichen.

Formate für die Zwischenablage

In Aufrufen von Zwischenablage-Funktionen werden die vordefinierten Windows-Formate durch Konstanten mit dem Präfix *cf_* angegeben. In Tabelle 16.1 sind die vordefinierten Formate aufgelistet.

Es handelt sich hier um Standardformate, die von vielen Windows-Anwendungen unterstützt werden. Es besteht die Möglichkeit, spezielle oder selbstdefinierte Formate zu registrieren. Daten, die in einem selbstdefinierten Format in der Zwischenablage gespeichert werden, können nur von Anwendungen geladen werden, die dieses Format kennen.

Jede Anwendung, die ein neues Format definiert, muß dieses Format durch einen Aufruf der Funktion *RegisterClipboardFormat* registrieren. Der erste Aufruf von *RegisterClipboardFormat* in einer Windows-Arbeitssitzung liefert einen neu-

en, eindeutigen Formatbezeichner für die Zwischenablage mit dem Präfix *cf_*. Alle Anwendungen, die zu einem späteren Zeitpunkt ein Format mit identischem Namen registrieren, erhalten denselben Formatbezeichner.

Tabelle 16.1 Formate für die Zwischenablage

Format	Bedeutung
cf_Text	Nullterminiertes Zeichen-Array.
cf_Bitmap	Bitmap-Format von Windows.
cf_Sylk	Symbolic-Link-Format von Microsoft.
cf_Tiff	Tag-image-file-Format.
cf_MetafilePict	Metafile-picture-Format von Windows.

Speichern von Daten in der Zwischenablage

Zur Speicherung von Daten in der Zwischenablage muß die Anwendung vier Schritte ausführen:

1. Öffnen der Zwischenablage (*OpenClipboard*).
2. Leeren der Zwischenablage (*EmptyClipboard*).
3. Speichern der neuen Daten in der Zwischenablage (*SetClipboardData*).
4. Schließen der Zwischenablage (*CloseClipboard*).

Ein Aufruf von *OpenClipboard* benötigt als Argument ein Handle auf ein Fenster. Dadurch wird sichergestellt, daß immer nur ein Fenster mit der Zwischenablage arbeiten kann.

Eine Anwendung, die Daten in der Zwischenablage speichern will, muß diese Daten in den globalen Speicher kopieren und in einem Aufruf der Windows-Funktion *SetClipboardData* ein Handle auf diese Daten übergeben (vergleichen Sie Kapitel 14 »Speicherverwaltung«). Nach der Übergabe des Handle gehören die Daten zur Zwischenablage, die Anwendung kann also nicht mehr mit ihnen arbeiten. Deshalb ist es wichtig, daß ein Handle auf eine *Kopie* der Daten übergeben wird.

Die Funktion *SetClipboardData* benötigt als Argument einen Zwischenablage-Formatbezeichner (zum Beispiel *cf_Text*). Die Anwendung kann *SetClipboardData* mehrmals aufrufen und dabei unterschiedliche Formate verwenden.

Das folgende Programm speichert Daten im *cf_Text*-Format in der Zwischenablage.

```
function MyWindow.CopyText(TextString: PChar) : Boolean;
var
  StringGlobalHandle: Handle;
  StringGlobalPtr: PChar;

begin
  CopyText := False;
```

```
    StringGlobalHandle := GlobalAlloc(gmem_Moveable,
                          StrLen(TextString) + 1);
    if StringGlobalHandle <> 0 then
    begin
      StringGlobalPtr := GlobalLock(StringGlobalHandle);
      if StringGlobalPtr <> nil then
      begin
        StrCopy(StringGlobalPtr, TextString);
        GlobalUnlock(StringGlobalHandle);
        if OpenClipboard(HWindow) <> 0 then
        begin
          EmptyClipboard;
          SetClipboardData(cf_Text, StringGlobalHandle);
          CloseClipboard;
          CopyText := True;
        end
        else GlobalFree(StringGlobalHandle);
      end
      else GlobalFree(StringGlobalHandle);
    end;
  end;
```

Die Textdaten der Anwendung – sie werden als *TextString*-Argument an *CopyText* übergeben – werden mit *StrCopy* zu *StringGlobalPtr* kopiert. Die in *StringGlobalPtr* gespeicherten Daten werden anschließend an *SetClipboardData* übergeben, somit kann *TextString* von der Anwendung weiter verwendet werden. Verwenden Sie bei der Reservierung des globalen Speichers mit *GlobalAlloc* nicht das Flag *gmem_Discardable*, da die Zwischenablage (sie ist am Ende der Eigentümer der Daten) die Daten nicht wiederherstellen kann.

Laden von Daten aus der Zwischenablage

Zum Laden von Daten aus der Zwischenablage muß die Anwendung vier Schritte ausführen:

1. Öffnen der Zwischenablage (*OpenClipboard*).

2. Feststellen, welche Formate sich in der Zwischenablage befinden (*IsClipboardFormatAvailable*).

3. Laden der Daten aus der Zwischenablage (*GetClipboardData*).

4. Schließen der Zwischenablage (*CloseClipboard*).

Das Laden von Daten aus der Zwischenablage erfolgt ebenso wie das Speichern mit Hilfe eines Handle auf den globalen Speicher. Da die Zwischenablage Eigentümer der Daten ist (und bleibt), muß die Anwendung die mit dem Handle verknüpften Daten kopieren. Dieses Handle sollte nicht direkt verwendet und manipuliert werden.

Das folgende Programm lädt Textdaten aus der Zwischenablage.

```
function MyWindow.PasteText(TextString: PChar;
                            TextSize: Integer) :
Integer;
var
  StringGlobalHandle: Handle;
  StringGlobalSize: Longint;
  StringGlobalPtr: PChar;
begin
  PasteText := -1;
  if OpenClipboard(HWindow) <> 0 then
  begin
    if IsClipboardFormatAvailable(cf_Text) <> 0 then
    begin
      StringGlobalHandle := GetClipboardData(cf_Text);
      if StringGlobalHandle <> 0 then
      begin
        StringGlobalSize := GlobalSize(StringGlobalHandle);
        StringGlobalPtr := GobalLock(StringGlobalHandle);
        if StringGlobalPtr <> nil then
        begin
          if TextSize < StringGlobalSize then
            StringGlobalSize := TextSize;
          StrLCopy(TextString, StringGobalPtr,
                   StringGlobalSize);
          GlobalUnlock(StringGlobalHandle);
          PasteText := StrLen(TextString);
        end;
      end;
    end;
    CloseClipboard;
  end;
end;
```

Eine Anwendung, die mit mehreren Datenformaten arbeiten kann, sollte zunächst das »informativste« Format anfordern. Enthält die Zwischenablage zum Beispiel ein binäres Format zur Übergabe von Daten, ist es besser, zuerst dieses Format anzufordern, da es beim *cf_Text*-Format durch Konvertierung von Zahlen in Text (und zurück) leicht zu Rundungsfehlern kommen kann.

Zeitverzögerte Datenkonvertierung

Die Ausführungsgeschwindigkeit von Anwendungen, die viele Datenformate unterstützen, würde sich beträchtlich verlangsamen, wenn die Daten bei jeder Speicherung in der Zwischenablage in alle Formate konvertiert werden müßten. Deshalb besteht die Möglichkeit, die Datenkonvertierung zeitlich solange hin-

auszuzögern, bis eine andere Anwendung Daten im betreffenden Format anfordert.

Übergeben Sie dazu für alle Formate, deren Konvertierung erst später erfolgen soll, im *SetClipboardData*-Aufruf 0 für das Handle im Parameter *Mem*. Eine Anwendung, die eine zeitverzögerte Datenkonvertierung initiiert, muß die zuletzt in der Zwischenablage gespeicherten Daten sichern, da sie erst später umgewandelt werden. Fordert eine andere Anwendung Daten in einem Format an, für das eine zeitverzögerte Umwandlung angegeben wurde, schickt Windows eine *wm_RenderFormat*-Botschaft an die Anwendung, die das zeitverzögerte Format in der Zwischenablage plaziert hat. Das angeforderte Format für die Daten wird im Word-Argument der *wm_RenderFormat*-Botschaft angegeben. Die Anwendung sollte dann die Daten im angeforderten Format in der Zwischenablage wie oben beschrieben speichern (mit dem Unterschied, daß die Zwischenablage nicht mehr geleert werden muß).

Die Anwendung, die die zeitverzögerte Konvertierung ausgelöst hat, erhält eine *wm_DestroyClipboard*-Botschaft, wenn die Zwischenablage von einer anderen Anwendung geleert wird. Sie kann dann die für eine spätere Konvertierung gespeicherten Daten löschen. Wird eine Anwendung mit zeitverzögerter Datenkonvertierung beendet, erhält sie von Windows eine *wm_RenderAllFormats*-Botschaft, damit sie die Datei in allen zeitverzögerten Formaten in der Zwischenablage speichern kann.

Botschaftsübermittlung zwischen Anwendungen

Die meisten Botschaften, die ein Fenster oder eine Anwendung empfängt, stammen von Windows oder von einem anderen Fenster derselben Anwendung (vgl. den Abschnitt »Anwenderdefinierte Botschaften« in Kapitel 7 »Überblick über ObjectWindows«). Es besteht aber auch die Möglichkeit, Botschaften von einer Anwendung zu einer anderen zu übermitteln. Dieser Vorgang ist im Protokoll für den Dynamischen Datenaustausch (DDE) formalisiert, Windows verfügt aber auch über einen Mechanismus für eine generalisierte Botschaftsübermittlung zwischen Anwendungen. Dieser Mechanismus eignet sich besonders zum Senden von Benachrichtigungsbotschaften und zur Übergabe einfacher Datenwerte. Zur Übermittlung komplexer Daten sollte jedoch das DDE-Protokoll eingesetzt werden.

Angenommen, Sie haben ein Programmpaket bestehend aus einer elektronischen Mailbox-Anwendung, einem Zeitplaner und einem Kalenderprogramm entwickelt. Die Mailbox-Anwendung enthält spezielle Routinen zum Abfangen von Benachrichtigungsbotschaften des Netz-Servers. Empfängt sie eine derartige Botschaft, muß sie die beiden anderen Programme davon in Kenntnis setzen. Dazu können Sie eine neue Botschaft mit dem Namen *wm_EMailArrived* definieren. Jede Anwendung, die diese neue Botschaft sendet oder empfängt, muß sie mit der Windows-Funktion *RegisterWindowMessage* registrieren. Verwenden Sie *SendMessage* oder *PostMessage*, um eine Botschaft von einer Anwendung an eine andere zu übermitteln. Soll jedes geöffnete Fenster die Botschaft

erhalten, übergeben Sie *HWnd(-1)* als Parameter für das Fenster-Handle. Zur Bestätigung des Erhalts kann die empfangende Anwendung eine weitere anwenderdefinierte Botschaft zurückschicken.

Im Longint-Parameter (*lParam*) darf kein Zeiger auf Daten übergeben werden, da Windows die Anwendung eventuell verlagert und der Zeiger dabei seine Gültigkeit verliert. Die Übergabe eines Handle auf den globalen Speicher ist ebenfalls unzulässig, da der betreffende Speicherblock neu reserviert, verschoben oder verworfen werden kann. Zu Problemen kann es auch kommen, wenn die sendende oder empfangende Anwendung das Speicher-Handle nicht freigibt. Übergeben Sie Daten, wenn möglich, immer direkt im Word- oder Longint-Parameter.

Dynamischer Datenaustausch

Der dynamische Datenaustausch (DDE) ist ein Protokoll, in dem die genauen Modalitäten für den Datentransfer zwischen zwei kooperierenden Anwendungen festgelegt sind. Die Benutzung des DDE-Protokolls setzt voraus, daß diese beiden Anwendungen genau wissen, in welcher Weise sie wechselseitig voneinander anwendungsspezifische Daten anfordern müssen. DDE ist in Form von Windows-Botschaften implementiert (sie beginnen alle mit *wm_DDE_*), die von einem Fenster der einen Anwendung zu einem Fenster der anderen Anwendung geschickt werden, und auf diese Weise Daten oder Anfragen übergeben.

Das DDE-Protokoll eliminiert außerdem Probleme, die beim Transfer globaler Daten auftreten können. DDE übergibt Textstrings als globale Atome und verwendet den globalen Speicher (er wird mit dem Flag *gmem_DDEShare* reserviert) für die Übergabe von Daten. Als Atome werden globale Stringkonstanten bezeichnet, auf die alle laufenden Windows-Anwendungen zugreifen können.

Begriffe

Wenn zwei Anwendungen gegenseitig Daten austauschen, führen sie miteinander eine *Konversation.* Ein einzelnes Fenster kann gleichzeitig nur an einer Konversation beteiligt sein. Eine Anwendung, die sich an mehreren Konversationen gleichzeitig beteiligen soll, muß mehrere Fenster erzeugen, von denen sich jeweils eines an einer Konversation beteiligt. Falls erforderlich, können diese zusätzlichen Fenster verborgen werden. Das Fenster, das die Konversation eröffnet, heißt Client-Fenster, das antwortende Fenster der anderen Anwendung wird als Server bezeichnet.

Das DDE-Protokoll verwendet für die Spezifikation der auszutauschenden Daten ein dreistufiges System. Die oberste Ebene bildet der Anwendungsname, gefolgt von dem Konversationsthema und dem Item-Namen. Der Anwendungsname ist wichtig, da jede Instanz einer Anwendung normalerweise einen speziellen Satz von Themen »versteht«. Ein Textverarbeitungsprogramm könnte zum Beispiel die Dateinamen von Dokumenten als Konversationsthema

benutzen. Als Item wird das angeforderte Datenelement bezeichnet. In einem Textverarbeitungsprogramm könnte dies ein spezieller Absatz oder eine bestimmte Schrift sein.

Anwendung und Thema werden zum Aufbau der Konversation benutzt, das Item ist die übertragene Dateneinheit. Die Daten können in jedem Zwischenablage-Format übertragen werden (vergleichen Sie Tabelle 16.1).

Aufbau einer Konversation

Nachdem der Client eine Konversation mit dem Server eröffnet hat, wird diese fortgeführt, bis sie vom Client oder vom Server wieder beendet wird.

Die Botschaften zum Eröffnen und zum Beenden einer Konversation lauten *wm_DDE_Initiate* bzw. *wm_DDE_Terminate.*

Um eine Konversation zu beginnen, schickt der Client mit der Funktion *SendMessage* eine *wm_DDE_Initiate*-Botschaft und verwendet dabei *HWnd(-1)* als Parameter für das Fenster-Handle. Da das DDE-Protokoll asynchron arbeitet, muß nur *wm_DDE_Initiate* mit *SendMessage* übergeben werden. Nachdem die Konversation eröffnet ist, werden mit der Funktion *PostMessage* weitere DDE-Botschaften übertragen. Im Gegensatz zu *SendMessage* kann bei *PostMessage* die sendende Anwendung fortlaufend Botschaften senden, ohne auf eine Antwort zu warten. *PostMessage* hat allerdings den Nachteil, daß die Client-Anwendung für das Abrufen erwarteter Botschaften verantwortlich ist.

Auf eine *wm_DDE_Initiate*-Botschaft können mehrere Fenster der Server-Anwendung antworten. Der Client muß den gewünschten Server auswählen und zu den anderen *wm_DDE_Terminate*-Botschaften senden, da immer nur ein Server mit einem Client Daten austauschen kann.

Wenn eine Anwendung eine Konversation mit *wm_DDE_Initiate* eröffnet, gibt sie den Namen der anderen Anwendung und das Konversationsthema an. Wird als Anwendungsname eine 0 angegeben, kann die Konversation mit jeder laufenden Anwendung geführt werden. Desgleichen kann für das Konversationsthema ein Null-Wert angegeben werden, um eine Konversation über jedes beliebige Thema zu ermöglichen.

Die folgende Methode eröffnet eine Konversation mit einem Fenster. Sie benötigt einen Anwendungsnamen und ein Konversationsthema als Argument. Sie könnten stattdessen auch Null-Werte verwenden, indem Sie nil angeben.

```
procedure TClientWindow.InitiateConversation(AppName, TopicName:
PChar);
var
  AppGobalAtom, TopicGlobalAddAtom: Word;
  lParam: Longint;
begin
  if AppName <> nil then
    AppGlobalAtom := GlobalAddAtom(AppName)
  else AppGlobalAtom := 0;
  if TopicName <> nil then
    TopicGlobalAtom := GlobalAddAtom(TopicName)
```

```
    else TopicGlobalAtom := 0;
    lParam := AppGlobalAtom or (TopicGlobalAtom shl 16);
    if AppGlobalAtom <> 0 then GlobalDeleteAtom(AppGlobalAtom);
    if TopicGlobalAtom <> 0 then GlobalDeleteAtom(TopicGlobalAtom);
  end;
```

In der Funktion *SendMessage* werden globale Atome anstelle von Zeigern übergeben. Die Atome werden nur dann erzeugt, wenn der String mit dem Anwendungsnamen oder dem Thema nicht nil ist. Die Anwendung muß (und kann) nach dem Aufruf von *SendMessage* alle erzeugten globalen Atome freigeben. Der Aufruf von *SendMessage* kehrt nämlich erst dann zurück, wenn alle Fenster im System die Möglichkeit hatten, auf die *wm_DDE_Initiate*-Botschaft zu reagieren. Die Richtlinien für die Erzeugung und Freigabe globaler Atome sind bei der Erläuterung der einzelnen DDE-Botschaften aufgeführt. Alle Anwendungsfenster, die den übergebenen Anwendungsnamen und das Thema bearbeiten können, antworten durch das Senden einer *wm_DDE_Ack*-Botschaft.

Zur Beantwortung einer *wm_DDE_Initiate*-Botschaft kann das Fensterobjekt einer Anwendung eine Botschaftsantwort-Methode definieren, die nach dem Empfang von *wm_DDE_Initiate* aufgerufen wird. Diese Methode muß das angegebene Thema überprüfen und ruft, falls eine Konversation möglich ist, eine weitere Methode auf, die dem Client-Fenster eine entsprechende Bestätigung übermittelt. Ist das Thema ein Null-Wert, hat das Server-Fenster die freie Wahl des Themas. Die Server-Anwendung erzeugt normalerweise ein neues Fenster zur Verwaltung der Konversation. Der Methode, die die Bestätigung übermittelt, muß ein Handle auf dieses neue Fenster übergeben werden.

Im folgenden Programm antwortet ein Server-Fenster namens *TestServer* auf eine *wm_DDE_Initiate*-Botschaft:

```
procedure TServerWindow.ACKTopic(ClientHWnd, ConversationHWnd: HWnd;
TopicName: PChar);
var
  AppGobalAtom, TopicGlobalAtom: Word;
  lParam: Longint;
begin
  AppGlobalAtom := GlobalAddAtom('TestServer');
  TopicGlobalAtom := GlobalAddAtom(TopicName);
  lParam := AppGlobalAtom or (TopicGlobalAtom shl 16);
  if SendMessage(ClientHWnd, wm_DDE_Ack, ConversationHWnd,
    lParam) = 0
    then
  begin
    GlobalDeleteAtom(AppGlobalAtom);
    GlobalDeleteAtom(TopicGlobalAtom);
  end;
end;
```

Anstatt die *HWindow*-Instanzvariable des Fensterobjekts *TServerWindow* zu verwenden, wird das Fenster, das die DDE-Konversation als Server verwaltet, als Parameter (*ConversationHWnd*) übergeben. So kann auch für jedes Thema ein neues Fensterobjekt erzeugt werden. Der Client kann dann wählen, mit wel-

chem Thema die Konversation fortgeführt werden soll. Beachten Sie, daß die globalen Atome erneut erzeugt und nur gelöscht werden, wenn *SendMessage* keinen Erfolg hat. Das Löschen der globalen Atome wird durch den Client vorgenommen, nachdem er die *wm_DDE_Ack*-Botschaft erhalten hat.

Die Konversation ist jetzt aufgebaut. Der Server kennt den Client, dessen Fenster-Handle er mit der *wm_DDE_Initiate*-Botschaft erhalten hat. Umgekehrt kennt der Client den Server, da er dessen Fenster-Handle über die *wm_DDE_Ack*-Botschaft erhalten hat. Diese Fenster-Handle werden in allen folgenden DDE-Botschaften verwendet, die zwischen den Anwendungen ausgetauscht werden.

Beenden einer Konversation

Die Konversation kann sowohl vom Client- als auch vom Server-Fenster durch das Senden einer *wm_DDE_Terminate*-Botschaft beendet werden. Normalerweise übernimmt der Client diese Aufgabe, nur beim Beenden seiner Anwendung schickt der Server die Botschaft. Nach Erhalt von *wm_DDE_Terminate* muß das empfangende Fenster eine *wm_DDE_Terminate*-Botschaft zurücksenden. Auf DDE-Botschaften, die nach der Verarbeitung einer *wm_DDE_Terminate*-Botschaft eingehen, darf nicht mehr geantwortet werden und alle globalen Atome oder Daten werden gelöscht. Nachdem das sendende Fenster die beantwortende *wm_DDE_Terminate*-Botschaft erhalten hat, kann es geschlossen werden.

wm_DDE_Terminate-Botschaften können auch unter extremen Bedingungen gesendet werden, etwa bei einem schweren Fehler, der die Beendigung einer Anwendung zur Folge hat. Es kann deshalb vorkommen, daß eine *wm_DDE_Terminate*-Botschaft nicht mehr beantwortet wird. Zum Schutz gegen derartige Fälle sollte die sendende Anwendung warten, bis sie eine *wm_DDE_Terminate*-Botschaft empfängt oder bis eine gewisse Zeit verstrichen ist. Alle DDE-Botschaften, die die *PostMessage*-Funktion verwenden, sollten über einen derartigen Timeout-Schutz verfügen. Vor einem Aufruf der *PostMessage*-Funktion kann auch *IsWindow* aufgerufen werden, um zu überprüfen, ob der Parameter für das Fenster-Handle des antwortenden Teils gültig ist. Diese Schutzmechanismen sind nicht Bestandteil des DDE-Protokolls, sollten aber eingebaut werden, um bei schweren Fehlern eine Fortsetzung der Anwendung zu ermöglichen.

Methoden für den Datenaustausch

Client und Server können auf fünf Arten miteinander kommunizieren:

1. Mit *wm_DDE_Request*- und *wm_DDE_Data*-Botschaften können einzelne Items übertragen werden.

2. Ein Client kann über die Änderung eines Items mit Hilfe von *wm_DDE_Advise*- und *wm_DDE_Data*-Botschaften informiert werden. Die Daten können anschließend mit den Botschaften *wm_DDE_Request*- und *wm_DDE_Data* abgerufen werden. Dieser Mechanismus liegt der Eigenschaft zugrunde, die als *Hot*

Link bezeichnet wird. Diese fortlaufende Aktualisierung kann mit *wm_DDE_-UnAdvise* beendet werden.

3. Die dritte Methode besteht aus einer Kombination von 1 und 2, wobei die Daten von *wm_DDE_Data*-Botschaften fortlaufend übertragen werden und somit eine *wm_DDE_Request*-Botschaft nicht erforderlich ist.

4. Eine *wm_DDE_Poke*-Botschaft weist den Server an, den Wert eines Items zu ändern.

5. Eine *wm_DDE_Execute*-Botschaft übergibt ein Befehlsmakro an die Server-Anwendung. Dieses Makro wird dann vom Server interpretiert und ausgeführt.

Anfordern eines einzelnen Items

Normalerweise fordert das Client-Fenster vom Server-Fenster Daten an. Mit dem folgenden Programm kann ein einzelnes Item angefordert werden:

```
procedure TClientWindow.RequestData(DataFormat: Word;
          Item: PChar; ServerHWnd: HWnd);
var
  ItemGlobalAtom: Word;
  lParam: Longint;
begin
  ItemGlobalAtom := GlobalAddAtom(Item);
  lParam := DataFormat or (ItemGlobalAtom shl 16);
  if PostMessage(ServerHWnd, wm_DDE_Request, HWindow, lParam) = 0
then
    GlobalDeleteAtom(ItemGlobalAtom);
end;
```

Der Parameter *DataFormat* enthält ein Format der Zwischenablage, zum Beispiel *cf_Text* oder *cf_DIF*. Kann der Server die Daten in das geforderte Format konvertieren, antwortet er mit einer *wm_DDE_Data*-Botschaft:

```
...
const
  DDE_FAckReq =      $8000;
  DDE_FDeferUpd =    $4000;
  DDE_FRelease =     $2000;
  DDE_FRequested =   $1000;

type
  TDDEData = record
    bitOptions: Word;
    cfFormat: Word;
  end;
...
```

```
procedure TServerWindow.ACKData(Item: PChar; DataFormat: Word; Data:
  Pointer; DataSize: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData) + DataSize);
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then GlobalFree(DataGlobalHandle)
    else
    begin
      DataRecord.cfFormat := DataFormat;
      DataRecord.bitOptions := DDE_FRequested or DDE_FAckReq;
      Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
      Move(Data, DataGlobalPtr + SizeOf(DataRecord), DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom := GlobalAddAtom(Item);
      lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
      if PostMessage(ClientHWnd, wm_DDE_Data, HWindow, lParam) = 0
        then
      begin
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      end
    end
  end;
end;
```

Einige Anmerkungen zu diesem Beispiel: Der globale Speicher wird als *gmem_- DDEShare* reserviert. Dadurch haben sowohl der Client als auch der Server Zugriff auf die durch das Handle bezeichneten Daten. Der Datenblock-Record steht im globalen Speicher vor den eigentlichen Daten. Auf diese Weise können das Format der Daten und einige Bit-Feld-Optionen (*DDE_FRequested* und *DDE_- FAckReq*) angegeben werden, die den Client anweisen, in einer bestimmten Art und Weise zu antworten.

DDE_FRequested bestimmt, daß die Botschaft *wm_DDE_Data* als Antwort auf eine *wm_DDE_FRequested*-Botschaft gesendet wird. *DDE_FAckReq* weist den empfangenden Client an, für die empfangene *wm_DDE_Data*-Botschaft eine *wm_DDE_FAckReq*-Bestätigungsbotschaft zu senden.

Wird als Datenformat *cf_Text* gewählt, muß jede Datenzeile im übergebenen Datenzeiger durch das Zeichenpaar Carriage-Return/Line-Feed beendet werden. Dies gilt auch für die letzte Zeile.

Kann der Server die *DDE_FRequested*-Botschaft nicht verarbeiten (wenn er

beispielsweise das Item nicht in das gewünschte Format konvertieren kann), muß er eine negative Bestätigung an den Client zurücksenden. Generell sollten Sie immer zuerst die komplexen Formate anfordern und sich dann zu den einfacheren »vortasten«, bis eine positive Bestätigung eintrifft. Die negative Bestätigung sieht folgendermaßen aus:

```
...
ItemGlobalAtom := GlobalAddAtom(Item);
lParam := ItemGlobalAtom shl 16;
if PostMessage(ClientHWnd, wm_DDE_Ack, HWindow, lParam) = 0 then
  GlobalDeleteAtom(ItemGlobalAtom);
...
```

Nach Erhalt der *wm_DDE_Data*-Botschaft kann der Client die eingehenden Daten verarbeiten. Er muß den *TDDEData*-Record am Anfang des Datenzeigers überprüfen und, wenn in *bitOptions DDE_FAckReq* enthalten ist, eine positive Bestätigung zurücksenden:

```
...
if ((DataRecord.bitOptions or DDE_FAckReq) <> 0) then
begin
  ItemGlobalAtom := GlobalAddAtom(Item);
  lParam := DDE_FAckReq or (ItemGlobalAtom shl 16);
  if PostMessage(ServerHWnd, wm_DDE_Ack, HWindow, lParam) = 0 then
    GlobalDeleteAtom(ItemGlobalAtom);
end;
...
```

Wenn die Bit-Option *DDE_FRelease* gesetzt ist, muß der Client das Speicher-Handle freigeben:

```
...
GlobalUnlock(ItemGlobalHandle);
if ((DataRecord.bitOptions or DDE_FRelease) <> 0) then
  GlobalFree(ItemGlobalHandle);
end;
...
```

In der *wm_DDE_Data*-Botschaft sollte auch das Flag *DDE_FAckReq* oder *DDE_FRelease* (oder beide) enthalten sein. Der globale Speicher wird entweder vom Server nach Erhalt einer Bestätigung oder vom Client freigegeben. Ist keines dieser Flags gesetzt, weiß der Server nicht, wann der globale Speicher freigegeben werden muß, und der Client wird ihn nicht freigeben.

Dieser Teil des DDE-Protokolls erfüllt drei wichtige Aufgaben:

1. Er stellt sicher, daß alle reservierten globalen Speicherobjekte freigegeben werden.

2. Er reserviert die globalen Speicherobjekte in einem Format, das sowohl vom Client als auch vom Server verarbeitet werden kann.

3. Er verhindert Deadlock-Situationen, in denen beide Anwendungen auf eine Antwort warten.

Andauernder Datentransfer

Ein Client kann eine andauernde Verbindung – ein Hot Link – einrichten, in der der Server Änderungen an einem Item meldet.

Eine derartige Verbindung wird durch das Senden einer *wm_DDE_Advise*-Botschaft eingerichtet:

```
procedure TClientWindow.Advise(Item: PChar; DataFormat: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Wort;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData));
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then GlobalFree(DataGlobalHandle)
    else
    begin
      DataRecord.cfFormat := DataFormat;
      DataRecord.bitOptions := DDE_FDeferUpd or DDE_FAckReq;
      Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom := GlobalAddAtom(Item)
      lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
      if PostMessage(ClientHWnd, wm_DDE_Advise, HWindow, lParam) = 0
        then
      begin
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      end
    end
  end;
end;
```

Wie Sie sehen, ist *DDE_FDeferUpd* im *bitOptions*-Feld von *TDDEData* gesetzt. Dadurch wird sichergestellt, daß alle folgenden *wm_DDE_Data*-Botschaften des Servers oder Client Daten enthalten. Die gesendete *wm_DDE_Data*-Botschaft ist mit der *ACKData*-Prozedur weiter oben identisch, das *DDE_FRequested*-Flag ist jedoch nicht gesetzt. Wenn das *DDE_FDeferUpd*-Feld nicht gesetzt ist, werden alle folgenden *wm_DDE_Data*-Botschaften ohne Daten gesendet, d.h. der Client muß die Daten mit *wm_DDE_Request* anfordern.

Ist das *DDE_FDeferUpd*-Flag gesetzt, könnte die Antwort folgendermaßen aussehen:

```
...
ItemGlobalAtom := GlobalAddAtom(Item);
lParam := ItemGlobalAtom shl 16;
  if PostMessage(ClientHWnd, wm_DDE_Data, HWindow, lParam) = 0 then
    GlobalDeleteAtom(ItemGlobalAtom);
...
```

Wenn der Server die Daten in das geforderte Format konvertieren kann, muß er eine positive *wm_DDE_Ack*-Bestätigung schicken. Kann er das nicht, muß er eine negative *wm_DDE_Ack*-Botschaft senden. Anschließend kann der Client versuchen, die Verbindung mit einem anderen Format der Zwischenablage aufzubauen.

Den Server zum Ändern eines Datenwertes auffordern

Mit der Botschaft *wm_DDE_Poke* kann der Server aufgefordert werden, den Wert eines Items zu ändern. Diese Botschaft wird folgendermaßen eingesetzt:

```
procedure TClientWindow.Poke(Item: PChar; DataFormat: Word; Data:
  Pointer; DataSize: Word);
var
  DataRecord: TDDEData;
  lParam: Longint;
  ItemGlobalAtom: Word;
  DataGlobalHandle: Wort;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    SizeOf(TDDEData) + DataSize);
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then
      GlobalFree(DataGlobalHandle)
    else begin
      DataRecord.cfFormat := DataFormat;
      Move(@DataRecord, DataGlobalPtr, SizeOf(DataRecord));
      Move(Data, DataGlobalPtr + SizeOf(DataRecord), DataSize);
      GlobalUnlock(DataGlobalHandle);
      ItemGlobalAtom := GlobalAddAtom(Item);
      lParam := DataGlobalHandle or (ItemGlobalAtom shl 16);
      if PostMessage(ServerHWnd, wm_DDE_Poke, HWindow, lParam) = 0
        then
      begin
        GlobalFree(DataGlobalHandle);
        GlobalDeleteAtom(ItemGlobalAtom);
      end
```

```
    end
  end;
end;
```

Der Code von *TClientWindow.Poke* weist eine starke Ähnlichkeit mit dem von *TServerWindow.ACKData* auf. Der Unterschied liegt im Adressaten: *wm_DDE_Poke* wird vom Client zum Server gesendet, während *wm_DDE_Data* vom Server zum Client geschickt wird. Kann der Server die *wm_DDE_Poke*-Botschaft verarbeiten, muß er eine positive, andernfalls eine negative Bestätigung senden. Der Server könnte bei der Verarbeitung der *wm_DDE_Poke*-Botschaft den Item-Namen zum Beispiel folgendermaßen erhalten:

```
...
var
  ItemName : array[0..128] of Char;
...
ItemGlobalAtom := lParam shr 16;
GlobalGetAtomName(ItemGlobalAtom, @ItemName, SizeOf(ItemName));
...
```

Ausführen von Makrobefehlen im Server

Mit der Botschaft *wm_DDE_Execute* kann der Client einen String an den Server senden, der als Befehlsmakro ausgeführt werden soll. Handelt es sich beispielsweise beim Server um ein Textverarbeitungsprogramm, muß der String das Format der Makrosprache des Textverarbeitungsprogramms haben. Der Server muß die Ausführung der *wm_DDE_Execute*-Botschaft durch eine positive oder negative Bestätigung zurückmelden. Das folgende Programm zeigt, wie diese Botschaft eingesetzt werden kann:

```
procedure TClientWindow.Execute(Command: Pointer; CommandSize: Word);
var
  lParam: Longint;
  DataGlobalHandle: Word;
  DataGlobalPtr: Pointer;
begin
  DataGlobalHandle := GlobalAlloc(gmem_Moveable or gmem_DDEShare,
    DataSize);
  if (DataGlobalHandle <> 0) then
  begin
    DataGlobalPtr := GlobalLock(DataGlobalHandle);
    if (DataGlobalPtr = nil) then
      GlobalFree(DataGlobalHandle)
    else begin
      Move(Data, DataGlobalPtr, DataSize);
      GlobalUnlock(DataGlobalHandle);
      lParam := DataGlobalHandle shl 16;
```

```
        if PostMessage(ServerHWnd, wm_DDE_Execute, HWindow, lParam) = 0
          then
        begin
          GlobalFree(DataGlobalHandle);
        end
      end
    end;
end;
```

Der gesendete String muß nullterminiert sein und folgende Syntax haben (die eckigen Klammern sind Teil des Befehls, die geschweiften Klammern enthalten optionale Elemente):

```
[macroCall]{[macroCall]}
```

Das Format von *macroCall* lautet:

```
macroName(parameter1 {, parameter2 ...})
```

Dazu ein Beispiel: Verfügt die Makrosprache Ihres Textverarbeitungsprogramms über die Fähigkeit, mit *goto_line(LineNo)* eine bestimmte Zeilennummer anzuspringen und den aktuellen Absatz mit *select_current_paragraph* auszuwählen, kann mit dem Befehlsstring [goto_line(50)][select_current_paragraph] ein Absatz gewählt werden, in der sich die Zeile 50 befindet.

In Anwendungen, die den dynamischen Datenaustausch als Server unterstützen *und* über eine umfangreiche Makrosprache verfügen, sind praktisch unendlich viele Interaktionen möglich.

Das Thema »System«

Alle Anwendungen, die DDE unterstützen, müssen eine spezielles Thema namens »System« verwalten. Clients, die mit Ihrer Anwendung nicht vertraut sind, können mit Hilfe dieses Themas feststellen, wie DDE in der Anwendung implementiert ist.

Das System-Thema sollte folgende Items unterstützen (Datenelemente innerhalb von Items müssen dabei durch Tabulatoren voneinander getrennt werden):

1. *SysItems*, eine Liste mit Items des System-Themas.

2. *Topics*, eine Liste mit allen momentan unterstützten Themen. Da einige Anwendungen für jede geöffnete Datei ein anderes Thema definieren, kann sich diese Liste ständig ändern.

3. *Formats*, eine Liste mit den Zwischenablage-Formaten, in die konvertiert werden kann.

17. Die grafische Geräteschnittstelle

In sehr vielen Windows-Anwendungen setzt sich die funktionsgerechte Anwenderschnittstelle aus Fenstern, Dialogen und Steuerelementen zusammen. Bei Zeichen- und Grafik-Anwendungen müssen Ihre Fenster Grafiken unterschiedlichster Natur aufnehmen können, zum Beispiel Linien, Formen, Text und/oder Bitmap-Bilder.

Die Grafikfähigkeit von Windows beruht auf einer Vielzahl von Funktionen, die unter dem Sammelbegriff *grafische Geräteschnittstelle* (*Graphics Device Interface*, abgekürzt *GDI*) zusammengefaßt werden. Sie können sich die GDI als Grafik-Engine vorstellen, die von Windows-Anwendungen für die Anzeige und Manipulation von Grafiken eingesetzt wird. Die Zeichenfähigkeit der GDI-Funktionen ist unabhängig von dem für die Anzeige verwendeten Ausgabegerät. So sind die Funktionen, die Sie für eine Anzeige auf einem EGA-Bildschirm einsetzen dieselben wie für die Ausgabe auf einem VGA-Bildschirm oder einem PostScript-Drucker.

Die Geräte-Unabhängigkeit wird durch Gerätetreiber erreicht, die die GDI-Funktionsaufrufe in Befehle übersetzen, die dem angeschlossenen Ausgabegerät entsprechen.

Bildschirmkontexte

Anders als bei herkömmlichen Grafikprogrammen unter DOS, ändert der Windows-Programmierer nie direkt die Pixel des Bildschirms, sondern führt Änderungen in einem virtuellen Bereich aus, den wir *Bildschirmkontext* nennen. Ein Bildschirmkontext ist eine virtuelle Oberfläche, die mit Attributen, wie Stift, Pinsel, Schrift, Hintergrundfarbe, Textfarbe und aktueller Position, verknüpft ist.

Wenn Sie zum Zeichnen eines Bildschirmkontexts GDI-Funktionen aufrufen, übersetzt der mit dem Kontext verknüpfte Gerätetreiber die Zeichenaktion in die entsprechenden Befehle. Diese Befehle setzen die Zeichenaktion so genau wie möglich für das jeweilige Ausgabegerät um. Es spielt keine Rolle, ob das Ausgabegerät ein Monochrom-Bildschirm oder ein Vier-Millionen-Farbenbildschirm ist.

Sie können sich den Bildschirmkontext als Leinwand für ein Ölgemälde

vorstellen, wobei das Fenster das Gemälde samt Rahmen darstellt. Anstatt direkt auf das gerahmte Bild zu zeichnen, zeichnen Sie auf eine Leinwand, die erst später als fertiges Gemälde auf einen Rahmen aufgezogen wird. Auf genau diese Weise zeichnen Sie auch Grafiken im Bildschirmkontext. Ein Bildschirmkontext beinhaltet *Zeichenwerkzeuge*, wie Stifte, Pinsel und Schrift.

Ein Bildschirmkontext ist ein von Windows verwaltetes Element, ähnlich wie ein Fensterelement, mit der Ausnahme, daß zum Bildschirmkontext kein korrespondierendes ObjectWindows-Objekt gehört.

Verwalten von Bildschirmkontexten

Damit Ihr Programm Grafiken in einen Bildschirmkontext zeichnen kann, muß für das gewünschte Fenster ein derartiger Kontext angefordert werden. Da der Bildschirmkontext sehr viel Speicherplatz benötigt, können während einer Windows-Sitzung nur fünf Bildschirmkontexte gleichzeitig zur Verfügung stehen. Das heißt, daß es nicht für jedes Fenster einen Bildschirmkontext geben kann. Ein Fenster fordert einen Bildschirmkontext nur dann an, wenn dieser benötigt wird. Der dabei belegte Speicherplatz muß aber schnellstmöglich wieder freigegeben werden.

Dieser Umstand sieht nun relativ beunruhigend aus, weil Sie keine Kontrolle über die Attribute des Bildschirmkontexts zu haben scheinen; ein anderes Fenster – sogar eine andere Anwendung – kann die Attribute ändern. Außerdem sind die Zeichenwerkzeuge nicht Bestandteile des Bildschirmkontext-Speichers. Sie müssen jedesmal für den benötigten Bildschirmkontext ausgewählt werden.

Wie Sie Bildschirmkontexte anfordern, verwenden und wieder freigeben, wird weiter hinten in diesem Kapitel im Abschnitt »Grafikanzeige in Fenstern« detailliert erklärt.

Inhalt des Bildschirmkontexts

Normalerweise werden Sie die Attribute des Bildschirmkontexts unverändert übernehmen. Trotzdem ist es wichtig zu wissen, was der Bildschirmkontext enthält.

Im vorliegenden Abschnitt erläutern wir kurz seine Eigenschaften, wie Bitmaps, Farben, Abbildungsmodi, Clipping-Fähigkeiten und Zeichenwerkzeuge. Auf einige dieser Punkte wird an anderen Stellen im Handbuch detaillierter eingegangen.

Bitmap-Grafiken

Die Oberfläche des Bildschirmkontexts wird als *Bitmap* bezeichnet. Bitmaps repräsentieren die Speicherkonfiguration eines bestimmten Ausgabegeräts. Aus diesem Grund sind Bitmaps abhängig von der Art des adressierten Ausgabegeräts. Darin liegt nun ein Problem: Bitmaps, die von einem Gerät gespeichert wurden, müssen nicht mit einem anderen Ausgabegerät kompatibel sein. GDI

kann diesem Problem jedoch mit Hilfe der sogenannten *geräteunabhängigen Bitmaps* begegnen. Zu den GDI-Funktionen, die Bitmaps erzeugen, gehören *CreateCompatibleDC*, *CreateCompatibleBitmap* und *CreateDIBitmap*. GDI-Funktionen, die Bitmaps manipulieren, umfassen *BitBlt*, *StretchBlt*, *StretchDIBits* und *SetDIBitsToDevice*.

Farbe

Die Farben, die ein Ausgabegerät für das Zeichnen verwendet, stehen in einer *Farbpalette*. Ist die gewünschte Farbe nicht in der Palette enthalten, können Sie diese hinzufügen. Genauer gesagt, erlauben Sie dem Gerätetreiber, die gewünschte Farbe durch Abwandlung anderer Farben in der Palette zu erzeugen. Farbpaletten werden im Abschnitt »Der Einsatz von Paletten« weiter hinten in diesem Kapitel erläutert.

Abbildungsmodi

Es ist schwierig Zeicheneinheiten auszuwählen, wenn Sie nicht wissen, welches Ausgabegerät für die Anzeige verwendet wird. Die meisten Anwendungen vernachlässigen dieses Problem einfach und gehen davon aus, daß die Standardeinheit (ein Pixel) ausreicht. Manche Anwendungen machen es jedoch unbedingt erforderlich, daß das gewünschte Bild exakt dargestellt wird. Für diese Ansprüche stellt die GDI verschiedene *Abbildungsmodi* zur Verfügung, die zum Teil geräteunabhängig sind.

Jeder Abbildungsmodus hat eine Einheit und eine Koordinatenorientierung. Der Standardabbildungsmodus setzt den Koordinaten-Ursprungspunkt auf die obere linke Ecke des Bildschirmkontexts, wobei die positive X-Achse nach rechts und die positive Y-Achse nach unten zeigen. Jeder Bildschirmkontext verfügt über ein Abbildungsmodus-Attribut, das festlegt, wie die angegebenen Koordinaten interpretiert werden müssen.

Es gibt Fälle, in denen eine Transformation zwischen den virtuellen Koordinaten, die Sie zum Zeichnen verwenden, und den physikalischen Bitmap-Koordinaten, in die der aktuelle Abbildungsmodus die virtuellen Koordinaten transformiert, erfolgen muß. Bei den meisten Anwendungen liegt der Ursprungspunkt für den Bildschirm in der oberen linken Ecke des Client-Bereichs. Einige Fenster rollen ihre Client-Oberfläche so, daß sich der Ursprungspunkt nicht mehr im Client-Bereich befindet. Einige GDI-Funktionen arbeiten in speziellen Koordinatensystemen, wodurch Konvertierungen erforderlich werden. Die GDI verfügt über Funktionen, die die notwendigen Transformationen ausführen: *ScreenToClient*, *ClientToScreen*, *DPToLP* und *LPToDP*.

Clipping-Regionen

Jeder Bildschirmkontext verfügt über ein Clipping-Regions-Attribut, das das Zeichnen außerhalb des zugelassenen Gebiets verhindert. Eine Clipping-Region kann ein komplexes Polygon oder eine elliptische Form sein, in denen die Zeichnungen auf der virtuellen Oberfläche des Bildschirmkontextes angezeigt werden.

Für die meisten Anwendungen reicht der Client-Bereich des Fensters als voreingestellte Clipping-Region aus. Nur Anwendungen, die spezielle optische Effekte produzieren, müssen eventuell die Clipping-Region ändern.

Zeichenwerkzeuge

Der Bildschirmkontext verwendet zum Zeichnen in erster Linie drei Zeichenwerkzeuge: *Stift*, *Pinsel* und *Schrift*. Der Stift wird zum Zeichnen von Linien, Kreisbögen und Linienzügen (Mehrfach-Liniensegmenten) verwendet. Zu den Attributen eines Stifts gehören Farbe, Breite und Art (zum Beispiel durchgezogen oder gepunktet). Ein Pinsel wird zum Ausfüllen von umrahmten Flächen, wie Rechtecken, abgerundeten Rechtecken, Ellipsen und Polygonen, eingesetzt. Funktionen, die umrahmte Flächen zeichnen, verwenden den Stift zum Zeichnen der Einrahmung und den Pinsel zum Ausfüllen der Fläche. Es gibt vier Pinselarten: kompakt, schraffiert, bitmap-gemustert und geräteunabhängig bitmap-gemustert. Das Schrift-Zeichenwerkzeug wird für das Zeichnen von Text auf dem Bildschirmkontext eingesetzt. Es legt Höhe, Gewicht, Zeichenabstand, Schriftfamilie und -art fest. Alle drei Zeichenwerkzeuge verwenden das Bildschirmkontext-Attribut der Hintergrundfarbe zum Ausfüllen der Abstände.

Zeichenwerkzeuge

Der Bildschirmkontext verwaltet die Anzeige der Grafiken auf dem Bildschirm. Um Grafiken auf verschiedene Art und Weisen darzustellen, können Sie die Zeichenwerkzeuge ändern. Die Attribute der Zeichenwerkzeuge bestimmen die Darstellungsform der mit den GDI-Funktionen (z.B. *LineTo*, *Rectangle* oder *TextOut*) erzeugten Grafiken. Stifte regeln die Darstellung von gezeichneten Linien, Pinsel das Aussehen von gefüllten Flächen und Schriften die Darstellung von gezeichnetem Text. Zur Zuweisung von Attributen an Zeichenwerkzeuge wählt das Windows-Programm ein virtuelles oder ein vordefiniertes Zeichenwerkzeug für den Bildschirmkontext aus. Ein virtuelles Zeichenwerkzeug wird von Ihrem Programm durch Belegen der Felder in einem bestimmten Record (*TLogPen*, *TLogBrush* oder *TLogFont*) erzeugt. Ein vordefiniertes Werkzeug ist bereits von Windows definiert und bietet die gebräuchlichsten Attribute – einen kompakten schwarzen Stift, einen grauen Pinsel sowie eine Systemschrift – an.

Vordefinierte Werkzeuge

Vordefinierte Zeichenwerkzeuge werden mit der GDI-Funktion *GetStockObject* erstellt.

Sehen Sie sich dazu folgendes Beispiel an:

```
var
  TheBrush: HBrush
begin
  TheBrush := GetStockObject(LtGray_Brush);
  ...
end;
```

LtGray_Brush ist eine Integerkonstante, die in der ObjectWindows-Unit *WinTypes* definiert ist. Die folgende Tabelle enthält die Konstanten aller verfügbaren vordefinierten Werkzeuge:

Tabelle 17.1 Vordefinierte Zeichenwerkzeuge

Pinsel	Stifte	Schriften
White_Brush	*White_Pen*	*OEM_Fixed_Font*
LtGray_Brush	*Black_Pen*	*ANSI_Fixed_Font*
Gray_Brush	*Null_Pen*	*ANSI_Var_Font*
DkGray_Brush		*System_Font*
Black_Brush		*Device_Default_Font*
Null_Brush		*System_Fixed_Font*
Hollow_Brush		

Vordefinierte Zeichenwerkzeuge sollten – im Gegensatz zu virtuellen – nach ihrem Gebrauch *nicht* gelöscht werden.

Virtuelle Zeichenwerkzeuge

Die Records für virtuelle Zeichenwerkzeuge, *TLogPen*, *TLogBrush* und *TLogFont* enthalten Felder für jedes Werkzeugattribut. *TLogPen.lopnColor* beinhaltet beispielsweise den Farbwert für den Stift. Jeder Recordtyp definiert seine eigenen, dem Werkzeugtyp entsprechenden Attribute.

Virtuelle Stifte

Mit den Windows-Funktionen *CreatePen* oder *CreatePenIndirect* können Sie virtuelle Stifte erzeugen:

```
ThePen := CreatePen(ps_Dot, 3, RGB(0, 0, 210));
ThePen := CreatePenIndirect(@ALogPen);
```

Die Definition des *TLogPen*-Records sieht folgendermaßen aus:

```
TLogPen = record
  lopnStyle: Word;
  lopnWidth: TPoint;
  lopnColor: Longint;
end;
```

Das Feld *lopnStyle* enthält eine Konstante für die Linienart (siehe Abb. 17.1).

Das Feld für die Breite, *lopnWidth*, enthält einen Punkt, dessen X-Koordinate ein Integerwert ist, der die Linienbreite in Geräteeinheiten angibt.

Für einen VGA-Bildschirm wird eine Linie mit einer Breite von 1 Pixel gezeichnet, wenn der Wert 0 ist. Der Wert für die Y-Koordinate wird ignoriert.

Das Farbfeld, *lopnColor* enthält einen Longint-Wert, dessen Bytes die Intensität der Grundfarben rot, grün und blau repräsentiert, die zur Kombination

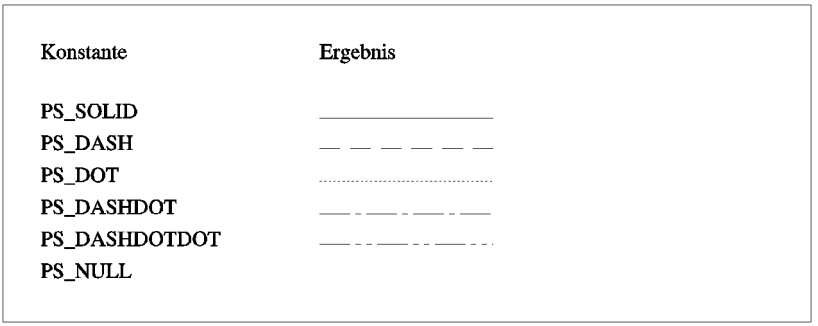


Abbildung 17.1 Linienarten für Stifte

anderer Farben herangezogen werden. Der Wert in *lopnColor* muß die Form $*00bbggrr* haben, wobei *bb* der Platzhalter für den Wert blau, *gg* für grün und *rr* für rot ist. Der zulässige Bereich für die Intensität der Grundfarben liegt zwischen 0 und 255, hexadezimal ausgedrückt zwischen 0 und FF.

In der folgenden Tabelle sehen Sie einige Beispiele für Farbwerte:

Tabelle 17.2 Beispiele für rot-grün-blaue Farbwerte

Wert	Farbe
$00000000	schwarz
$00FFFFFF	weiß
$000000FF	rot
$0000FF00	grün
$00FF0000	blau
$00808080	grau

Alternativ dazu können Sie mit der *RGB*-Funktion Farben erzeugen. Zum Beispiel liefert *RGB(0,0,0)* schwarz, *RGB(255,0,0)* rot, etc.

Virtuelle Pinsel

Mit den Windows-Funktionen *CreateHatchBrush*, *CreatePatternBrush*, *CreateDIBPatternBrush* oder *CreateBrushIndirect* werden virtuelle Pinsel erzeugt, zum Beispiel:

```
TheBrush := CreateHatchBrush(hs_Vertical, RGB(0, 255, 0));
TheBrush := CreateBrushIndirect(@ALogBrush);
```

Die Definition des *TLogBrush*-Records sieht folgendermaßen aus:

```
TLogBrush = record
  lbStyle: Word;
  lbColor: Longint;
  lbHatch: Integer;
end;
```

Das Feld *lbStyle* enthält eine Integerkonstante, die die Pinselart anzeigt:

bs_DIBPattern besagt, daß der Musterpinsel von einem geräteunabhängigen Bitmap definiert ist.

bs_Hatched legt eines der vordefinierten Schraffurmuster fest (vgl. *lbHatch*).

bs_Hollow ist ein leerer Pinsel.

bs_Pattern verwendet das Quadrat mit 8 x 8 Pixel in der oberen linken Ecke des Bitmaps, das aktuell im Speicher steht.

bs_Solid ist ein kompakter Pinsel.

Das *lbColor*-Feld beinhaltet einen Farbwert, wie er für die *TLogPen*-Records erforderlich ist. Das Feld wird von Pinseln mit den Stilen *bs_Hollow* und bs_*Pattern* ignoriert.

Das *lbHatch*-Feld enthält eine Integerkonstante, die das Schraffurmuster für Pinsel mit dem Stil bs_*Hatches* angibt. Wenn der Stil *bs_DIBPattern* ist, enthält *lbHatch* ein Handle auf das Bitmap.

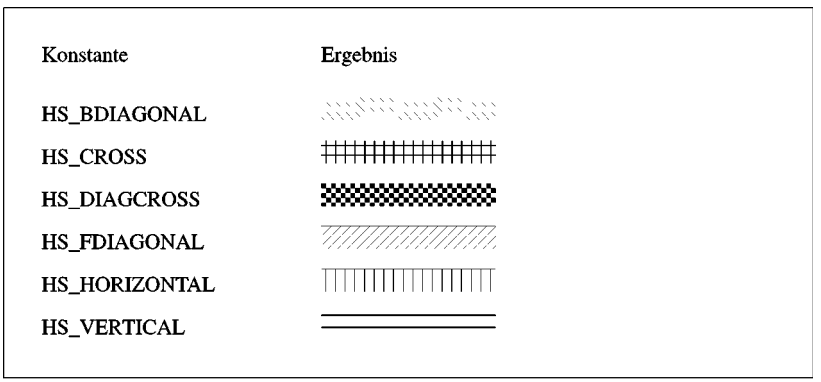


Abbildung 17.2 Schraffurmuster für Pinsel

Virtuelle Schriften

Mit den Windows-Funktionen *CreateFont* oder *CreateFontIndirect* können Sie virtuelle Schriften erzeugen.

Die Definition des *TLogFont*-Records sieht folgendermaßen aus:

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
```

```
    lfQuality: Byte;
    lfPitchAndFamily: Byte;
    lfFaceName: array[0..lf_FaceSize - 1] of Byte;
  end;
```

Zur Erstellung eines Schrift-Werkzeugs mit *TLogFont* geben Sie die Attribute der gewünschten Schriften an.

Ihr Programm übernimmt diese Informationen allerdings nicht und erzeugt aus der verfügbaren Information eine Bildschirmschrift, sondern es bildet die Schriftanforderung auf eine aktuell in der Windows-Sitzung definierte Bildschirmschrift ab.

Das *lfHeight*-Feld legt die gewünschte Höhe der Schriften fest. Bei der Angabe von 0 wird eine Standardgröße herangezogen. Ein positiver Wert ist die Zellenhöhe in virtuellen Einheiten. Ein negativer Wert wird in einen positiven Wert umgewandelt und ergibt die Zeichenhöhe in virtuellen Einheiten.

Das Feld *lfWidth* gibt die gewünschte Zeichenbreite in Geräteeinheiten an. Wenn der Wert 0 ist, wird sie entsprechend dem Höhen- und Seitenverhältnis des Bildschirms berechnet.

Für gedrehten Text setzt *lfEscapement* einen Wert in Zehntel Grad, um den der Text gegen den Uhrzeigersinn gedreht wird. *lfOrientation* hat dieselbe Funktion für die Drehung einzelner Zeichen.

Das gewünschte Gewicht wird mit *lfWeight* festgelegt. Dazu stehen Ihnen die Konstanten *fw_Light*, *fw_Normal*, *fw_Bold* und *fw_DontCare* zur Verfügung.

Wenn in *lfItalic*, *lfUnderline* und *lfStrikeOut* keine Null-Werte stehen, werden die entsprechenden Schriftattribute – kursiv, unterstrichen sowie durchgestrichen – vergeben.

Das Feld *lfCharSet* fordert bestimmte Zeichensätze, wie *ANSI_CharSet*, *OEM_CharSet* oder *Symbol_CharSet*, an.

Die ANSI-Zeichen finden Sie in Anhang B des Microsoft-Windows-Handbuches. *OEM_CharSet* ist systemabhängig.

Das Feld *lfOutPrecision* bestimmt, wie genau die von Windows zur Verfügung gestellte Schrift mit der angeforderten Größe und Position übereinstimmen muß. Voreingestellt ist *Out_Default_Precis*.

Das Feld *lfClipPrecision* legt fest, wie teilweise sichtbare Zeichen abgeschnitten werden. Voreingestellt ist *Clip_Default_Precis*.

Das Feld *lfQuality* zeigt an, wie genau die von Windows zur Verfügung gestellte Schrift mit den angeforderten Schriftattributen übereinstimmen soll. Das Feld kann mit *Default_Quality*, *Draft_Quality* oder *Proof_Quality* belegt werden.

Bei *Proof_Quality* werden fette, kursive, unterstrichene und durchgestrichene Schriften künstlich erzeugt, falls sie nicht verfügbar sind.

Das Feld *lfPitchAndFamily* fordert einen Zeichenabstand und eine Schriftfamilie an. Es kann sich aus einer or-Operation zwischen einer Zeichenabstand- und einer Familienkonstante ergeben (siehe Tabelle 17.3).

lfFaceName ist ein String, der die erforderliche Schriftart festlegt. Wenn der Wert 0 ist, wird eine Schriftart verwendet, die auf den Werten in den anderen *TLogFont*-Feldern basiert.

Tabelle 17.3 Zeichenabstand- und Familienkonstanten für Schriften

Zeichenabstand	Familie
Default_Pitch	*ff_Modern*
Fixed_Pitch	*ff_Roman*
Variable_Pitch	*ff_Script*
	ff_Swiss
	ff_Decorative
	ff_DontCare

Im folgenden finden Sie einige Beispielschriften mit dem Quellcode, der die *TLogFont*-Records definiert hat;

Sample Text

```
procedure MyWindow.MakeFont;
var
  MyLogFont: TLogFont;
begin
  with MyLogFont do
  begin
    lfHeight := 20;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Bold;
    lfItalic := 0;
    lfUnderline := 0;
    lfStrikeOut := 0;
    lfCharSet := ANSI_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Default_Quality;
    lfPitchAndFamily : Variable_Pitch or ff_Swiss;
    StrCopy(@lfFaceName, 'Helv');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Sample Text

```
procedure MyWindow.MakeFont;
var
  MyLogFont: TLogFont;
begin
  with MyLogFont do
```

```
  begin
    lfHeight := 10;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Normal;
    lfItalic := Ord(True);
    lfUnderline := Ord(True);
    lfStrikeOut := 0;
    lfCharSet := ANSI_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Default_Quality;
    lfPitchAndFamily : Fixed_Pitch or ff_DontCare;
    StrCopy(@lfFaceName, 'Courier');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Σαμπλε Τεξτ

```
procedure MyWindow.MakeFont;
var
  MyLogFont: TLogFont;
begin
  with MyLogFont do
  begin
    lfHeight := 20;
    lfWidth := 0;
    lfEscapement := 0;
    lfOrientation := 0;
    lfWeight := fw_Normal;
    lfItalic := 0;
    lfUnderline := 0;
    lfStrikeOut := 0;
    lfCharSet := Symbol_CharSet;
    lfOutPrecision := Out_Default_Precis;
    lfClipPrecision := Clip_Default_Precis;
    lfQuality := Proof_Quality;
    lfPitchAndFamily : Variable_Pitch or ff_Roman;
    StrCopy(@lfFaceName, 'Tms Rmn');
  end;
  TheFont := CreateFontIndirect(@MyLogFont);
end;
```

Grafikanzeige in Fenstern

Eine Windows-Anwendung sorgt für das Neuzeichnen ihrer Fenster, wenn sie zum ersten Mal angezeigt oder aktualisiert werden müssen, zum Beispiel wenn sie nach der Darstellung als Symbol wiederhergestellt oder von einem anderen sie überdeckenden Fenster freigegeben werden.

Da Windows kein automatisches Neuzeichnen von Fensterinhalten vorsieht, muß das Fenster benachrichtigt werden, wenn es sich selbst neuzeichnen soll. Im vorliegenden Abschnitt erfahren Sie, wie in einem Fenster gezeichnet wird, wie der Aktualisierungsmechanismus abläuft und wie der Bildschirmkontext verwendet wird.

Im folgenden werden mit *Grafiken* sowohl Text-, als auch grafische Elemente bezeichnet, wie beispielsweise Bitmaps und Rechtecke.

Zeichnen in Fenstern

Um Grafiken in einem Fensterobjekt zeichnen zu können, müssen Sie einen Bildschirmkontext *anfordern*, der nach dem Zeichenvorgang *freigegeben* werden muß. (Während einer Windows-Sitzung stehen maximal fünf Bildschirmkontexte zur Verfügung.) In der Zwischenzeit können Sie das Bildschirmkontext-Handle als Argument in jeder Windows-Grafikfunktion einsetzen.

Verwalten eines Bildschirmkontexts

Normalerweise werden Sie ein Fensterobjektfeld definieren, das das Handle auf den aktuellen Bildschirmkontext enthält, vergleichbar mit *HWindow*, in dem ein Handle auf ein Fenster gespeichert ist:

```
type
  TMyWindow = object(TWindow)
    TheDC: HDC;
    ...
end;
```

Wenn Sie für ein Fenster einen Bildschirmkontext anfordern wollen, rufen Sie die Windows-Funktion *GetDC* auf:

```
TheDC := GetDC(HWindow);
```

Danach können Sie Zeichenoperationen im Bildschirmkontext ausführen. Sie können das Bildschirmkontext-Handle in Windows-Grafikfunktionen verwenden:

```
LineTo(TheDC, Msg.LParamLo, Msg.LParamHi);
```

Sobald Sie die Arbeit mit dem Bildschirmkontext beendet haben, geben Sie ihn mit dem Aufruf der Funktion *ReleaseDC* frei:

```
ReleaseDC(HWindow, TheDC);
```

Zwischen zwei aufeinanderfolgenden Aufrufen von *GetDC* muß ein *ReleaseDC*-Aufruf stehen. Andernfalls kann es während des Programmablaufs zu einem Systemfehler kommen, weil der für die Bildschirmkontexte zur Verfügung stehende Speicherplatz nicht mehr ausreicht.

Aufrufen von Grafikfunktionen in Fenstern

GDI-Funktionen benötigen ein Handle auf den zu bearbeitenden Bildschirmkontext. Normalerweise werden Sie diese Funktionen innerhalb der Methoden eines Fenstertyps aufrufen. So ist *TextOut* zum Beispiel eine Funktion, die Text an der angegebenen Stelle in einem Bildschirmkontext zeichnet:

```
TheDC := GetDC(HWindow);
TextOut(TheDC, 50, 50, 'Sample Text', 11);
ReleaseDC(HWindow, TheDC);
```

Aktualisieren von Fenstern

Wenn ein Fenster erneut gezeichnet werden muß, ist es *ungültig* geworden, d.h. seine Anzeige ist nicht länger korrekt und muß aktualisiert werden. Dies ist der Fall, wenn ein Fenster zum ersten Mal angezeigt wird, wenn es nach der Verkleinerung auf ein Symbol wiederhergestellt wird oder wenn ein anderes Fenster so verschoben wird, daß Teile des dahinterliegenden Fensters sichtbar werden. In all diesen Fällen sendet Windows eine *wm_Paint*-Botschaft an die entsprechende Anwendung. Als Reaktion auf dieser Botschaft wird die *Paint*-Methode des Fensters aufgerufen. Einer der *Paint*-Parameter, *PaintDC*, ist der Bildschirmkontext, der für die Aktualisierung verwendet wird. Damit erspart man sich das Anfordern eines Bildschirmkontexts innerhalb der Paint-Methode; er steht mit *PaintDC* bereits zur Verfügung. Die *Paint*-Methode von *TWindow* zeichnet nichts, da *TWindow*-Objekte keine Grafiken beinhalten. Definieren Sie daher in Ihren Fenstertypen eine *Paint*-Methode, die Methoden und Funktionen zum Zeichnen von Grafiken und Text aufruft.

Mit Hilfe des Bildschirmkontexts können Sie neue Zeichenwerkzeuge für den Kontext auswählen, wie beispielsweise Stifte mit verschiedenen Farben oder Pinsel mit verschiedenen Mustern. Sie müssen diese Werkzeuge erneut über die *Paint*-Methode in den Bildschirmkontext aufnehmen. Zum Abschluß der *Paint*-Methode wird der benutzte Bildschirmkontext freigegeben.

Grafik-Strategie

Die *Paint*-Methode sorgt dafür, daß die aktuellen Fensterinhalte jederzeit neugezeichnet werden können. Dazu gehört auch die erste Anzeige eines Fensters. Aus diesem Grund muß die *Paint*-Methode alle »permanenten« Grafiken des Fensters zeichnen können. Zusätzlich sollte sie Grafiken, die seit der ersten Anzeige des Fensters hinzugekommen sind, aktualisieren können. Zur

Produktion dieser »dynamischen« Grafiken muß die *Paint*-Methode auf die Befehle oder Daten zugreifen, mit denen die Grafiken erstellt wurden.

Um dies zu erreichen, haben Sie die Wahl zwischen zwei Möglichkeiten. Die erste Möglichkeit besteht darin, daß Sie zuerst den grafikerzeugenden Code in einige wenige Methoden zerlegen und diese dann von der *Paint*-Methode aus aufrufen, wenn Grafiken dynamisch erstellt werden müssen. Der zweite Weg liegt – wie in dem Beispiel in Kapitel 3 »Füllen eines Fensters« ersichtlich – darin, die den Grafikinhalt eines Fensters betreffenden Daten in einem Feld des entsprechenden Fensterobjekts zu speichern. Zu diesen Daten gehören beispielsweise Koordinaten, Formeln und Bitmaps. Spielen Sie dann in der *Paint*-Methode die Grafikroutinen ab, die für die Transformierung der Daten in Grafiken erforderlich sind. Mit dieser Strategie und der Fähigkeit von Objekten, die eigenen Daten und Funktionen speichern zu können, steht der Produktion von »lebendigen« Grafikanwendungen nichts mehr im Wege.

Die Verwendung von Zeichenwerkzeugen

Der Bildschirmkontext erlaubt Ihnen nicht nur das Zeichnen in einem Fenster, sondern er stellt auch die dazu notwendigen Werkzeuge zur Verfügung: Stifte, Pinsel, Schriften und Paletten. Wenn Sie im Bildschirmkontext eine Linie zeichnen, wird die Linie gemäß den Attributen des aktuellen Stifts erzeugt. Dazu gehören Farbe, Art (durchgezogen, gepunktet, etc.) und Dicke. Das Ausfüllen eines Bereichs richtet sich nach den Attributen des aktuellen Pinsels, also nach seinem Füllmuster und seiner Farbe. Wenn Sie Text in einem Bildschirmkontext zeichnen, erscheint er mit der Schrift (Modern, Roman, Swiss, etc.), der Größe und der Art (kursiv, fett, etc.) des aktuellen Schrift-Werkzeugs. Eine Palette enthält eine Kollektion mit den aktuell verfügbaren Farben.

In einem Bildschirmkontext gibt es jeweils ein Zeichenwerkzeug von jedem Typ. Ein neu zugewiesener Bildschirmkontext enthält die voreingestellten Werkzeuge: einen dünnen schwarzen Stift, einen kompakten schwarzen Pinsel, eine Systemschrift und eine Standardpalette. Wenn Ihnen diese Werkzeuge ausreichen, können Sie sie unverändert übernehmen.

Falls Sie die voreingestellten Werkzeuge ändern möchten, müssen Sie ein neues Werkzeugelement erstellen und für den Bildschirmkontext auswählen. Wenn Sie zum Beispiel einen neuen Stift wählen, wird der alte Stift deaktiviert. Sie sollten die vorhergehenden Werkzeuge speichern und sie wieder auswählen, wenn Sie die Arbeit mit dem neuen Werkzeug beendet haben:

```
var
  newPen, OldPen: HPen;
  TheDC: HDC;
begin
  { erzeugt eine Schrift mit der Breite 10 }
  NewPen := CreatePen(ps_Solid, 10, RGB(0, 0, 0));
  TheDC := GetDCCAWindow^.HWindow);
  OldPen := SelectObject(TheDC, NewPen);
  { hier wird gezeichnet }
```

```
    SelectObject(TheDC, OldPen);
    ReleaseDC(AWindow^.HWindow, TheDC);
    DeleteObject(NewPen);
  end;
```

Wie dieses Beispiel deutlich macht, müssen neue Zeichenwerkzeuge erstellt und dann gelöscht werden. Zeichenwerkzeuge sind wie Bildschirmkontexte als Elemente im Windows-Speicher abgelegt. Werden die Werkzeuge nicht gelöscht, kommt es zu Speicherverlust und eventuell Fehlern. Ebenso wie bei den Bildschirmkontexten sollten Sie Handles auf Zeichenwerkzeuge von unterschiedlichen Typen speichern: *HPen*, *HBrush*, *HFont* und *HPalette*.

Die Windows-Funktion *DeleteObject* entfernt Zeichenwerkzeuge aus dem Windows-Speicher. *Löschen* Sie aber *nicht* die Zeichenwerkzeuge, die aktuell für einen Bildschirmkontext ausgewählt sind.

Obwohl ein Bildschirmkontext immer nur ein Zeichenwerkzeug desselben Typs enthalten kann, können Sie ein »Lager« mit verfügbaren Werkzeugen anlegen. Wichtig ist nur, daß Sie alle Werkzeuge wieder löschen, wenn Ihre Anwendung beendet wird. Sie können dabei so vorgehen, wie es in dem Beispiel in Kapitel 3 »Füllen eines Fensters« gezeigt wird: Definieren Sie ein Fensterobjektfeld mit dem Namen *ThePen*, um darin ein Handle auf das aktuelle Stift-Werkzeug zu speichern. Wenn der Anwender eine neue Stiftart auswählt, wird ein neues Stift-Werkzeug erstellt und das alte gelöscht. Der zuletzt aktive Stift wird mit der Hauptfenstermethode *CanClose* gelöscht. Die voreingestellten Werkzeuge, die ein neu zugewiesener Bildschirmkontext enthält, müssen Sie nicht löschen.

Es gibt zwei Möglichkeiten zur Erstellung von neuen Zeichenwerkzeugen. Der einfachste Weg ist die Verwendung eines bereits existierenden, *vordefinierten* Werkzeugs. Die vordefinierten Werkzeuge finden Sie in Tabelle 17.1.

Wenn Sie ein vordefiniertes Werkzeug in den Bildschirmkontext einbringen wollen, verwenden Sie die Methoden *SetStockPen*, *SetStockBrush*, *SetStockFont* und *SetStocKpalette*. Sehen Sie sich das folgende Beispiel an:

```
ThePen := GetStockObject(Black_Pen);
```

Wichtig: Löschen Sie die vordefinierten Werkzeuge nicht aus dem Windows-Speicher.

Es kann vorkommen, daß es kein vordefiniertes Werkzeug gibt, das über die von Ihnen gewünschten Attribute verfügt. So erzeugen alle vordefinierten Stift-Werkzeuge nur dünne schwarze Linien. Wenn Sie nun aber dicke Linien zeichnen wollen, haben Sie zwei Möglichkeiten zur Erstellung des gewünschten Werkzeugs.

Sie können die Windows-Funktionen *CreatePen*, *CreateFont*, *CreateSolidBrush* oder *CreateDIBPatternBrush* aufrufen. Diese Funktionen haben Parameter, die das gewünschte Werkzeug beschreiben und Werkzeug-Handles zurückliefern, die dann in den *SelectObject*-Aufrufen eingesetzt werden können.

Die zweite Möglichkeit zum Erstellen eigener Werkzeuge besteht darin, daß Sie eine Beschreibung für die Attribute eines *virtuellen* Werkzeugs aufbauen. Ein virtuelles Werkzeug ist in den Windows-Datenstrukturen *TLogPen*, *TLogBrush*, *TLogFont* und *TLogPalette* enthalten. So verfügt zum Beispiel *TLogPen* über Felder, in denen Breite, Farbe und Art vermerkt sind. Sobald Sie eine Datenstruk-

tur für das virtuelle Werkzeug definiert haben, übergeben Sie diese als Parameter an *CreatePenIndirect*, *CreateBrushIndirect*, *CreateFontIndirect* oder *CreatePalette*. Diese Funktionen liefern Werkzeug-Handle zurück, die in *SelectObject*-Aufrufen verwendet werden können. Das folgende Beispiel setzt den Stift eines Bildschirmkontexts auf die Farbe blau:

```
procedure SampleWindow.ChangePenToBlue;
var
  ALogPen: TLogPen;
  ThePen: HPen;
begin
  ALogPen.lopnColor := RGB(0, 0, 255);
  ALogPen.lopnStyle := ps_Solid;
  ALogPen.lopnWidth.X := 0;
  ALogPen.lopnWidth.Y := 0;
  Thepen := CreatePenIndirect(@ALogPen);
  SelectObject(TheDC, ThePen);
end;
```

GDI-Zeichenfunktionen

In diesem Abschnitt werden die verschiedenen API-Aufrufe erläutert, die für das Zeichnen in Fenstern verfügbar sind.

Funktionen für das Zeichnen von Text

Die Funktionen für das Zeichnen von Text verwenden die aktuelle Schrift des angegebenen Bildschirmkontexts. Die Funktion *TextOut* zeichnet Text an dem angegebenen Punkt. *TextOut* richtet sich bei der Textausrichtung nach dem Wert des aktuellen Textformat-Flags, das standardmäßig linksbündig ausrichtet. Die aktuell gültige Ausrichtung kann mit der Funktion *GetTextAlign* ermittelt und mit der Funktion *SetTextAlign* festgelegt werden.

Die *TextOut*-Funktion ist die am häufigsten verwendete Funktion zum Zeichnen von Text. Die folgende *Paint*-Methode zeichnet, unter Verwendung des voreingestellten Textformat-Flags, ein links ausgerichtetes Zeichen-Array, dessen obere linke Ecke an der Position (10,15) liegt.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  MyTextString: array[0..20] of Char;
begin
  StrCopy(MyTextString, 'Hello, World');
  TextOut(PaintDC, 10, 15, MyTextString, StrLen(MyTextString));
end;
```

Abbildung 17.3 Das Ergebnis der Funktion TextOut

Funktionen für das Zeichnen von Linien

Die Funktionen für das Zeichnen von Linien verwenden den aktuellen Stift des angegebenen Bildschirmkontexts. Die überwiegende Mehrzahl aller Linien wird mit den Funktionen *MoveTo* und *LineTo* gezeichnet. Diese Funktionen betreffen das Bildschirmkontext-Attribut aktuelle Position. Die aktuelle Position entspricht beim Zeichnen mit Papier und Bleistift dem Punkt, an dem der Bleistift auf das Papier aufgesetzt wird.

MoveTo und LineTo

Die Funktion *MoveTo* verschiebt die aktuelle Position an den angegebenen Koordinatenpunkt. Die Funktion *LineTo* zeichnet eine Linie mit dem Stift von der aktuellen Position zu dem angegebenen Koordinatenpunkt. Die aktuelle Position wird anschließend entsprechend aktualisiert. Die folgende *Paint*-Methode zeichnet eine Linie von (100, 150) nach (10, 15):

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
begin
  MoveTo(PaintDC, 100, 150);
  LineTo(PaintDC, 10, 15);
end;
```

Abbildung 17.4 Das Ergebnis der Funktion LineTo

Polyline

Die Funktion *Polyline* zeichnet mehrere Linien durch Verbinden von Punkten. Ähnliches würde erreicht, wenn zuerst die Funktion *MoveTo* und danach

mehrmals die Funktion *LineTo* aufgerufen würde. *Polyline* führt die Operation jedoch erheblich schneller aus und läßt die aktuelle Position unverändert. Die folgende *Paint*-Methode zeichnet einen rechten Winkel:

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo:
TPaintStruct);
var
  Points: array[0..2] of TPoint;
begin
  Points[0].X := 10;
  Points[0].Y := 15;
  Points[1].X := 10;
  Points[1].Y := 150;
  Points[2].X := 100;
  Points[2].Y := 150;
  Polyline(PaintDC, @Points, 3);
end;
```

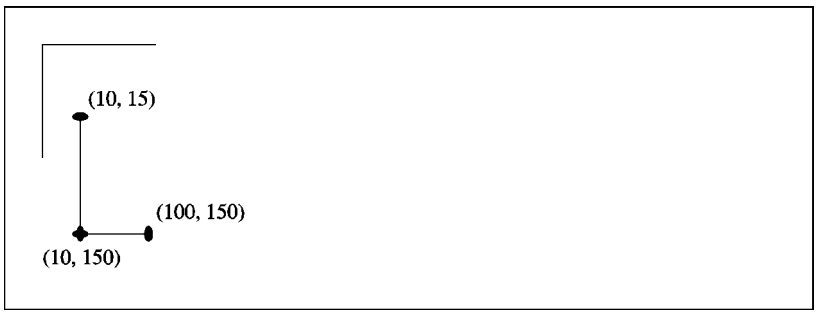


Abbildung 17.5 Das Ergebnis der Funktion Polyline

Arc

Die Funktion *Arc* zeichnet einen Kreisbogen mit dem Umfang der Ellipse, die von dem angegebenen Rechteck umgeben wird. Der Kreisbogen beginnt am Schnittpunkt der Ellipsenecke mit der Linie, die vom Mittelpunkt zum angegebenen Startpunkt führt. Der Bogen wird gegen den Uhrzeigersinn gezeichnet, bis er den Punkt erreicht, an dem die Ellipse sich mit der Linie schneidet, die vom Mittelpunkt der Ellipse zum angegebenen Endpunkt führt.

Die folgende *Paint*-Methode zeichnet das obere Viertel eines Kreises, beginnend bei (40, 25) und endend bei (10, 25). Dazu wird das umgebende Rechteck (10, 10), der Startpunkt (0, 0) und der Endpunkt (50, 0) verwendet. Das ist möglich, auch wenn der angegebene Start- und Endpunkt nicht auf dem Kreisbogen liegen.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo:
TPaintStruct);
begin
  Arc(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
end;
```

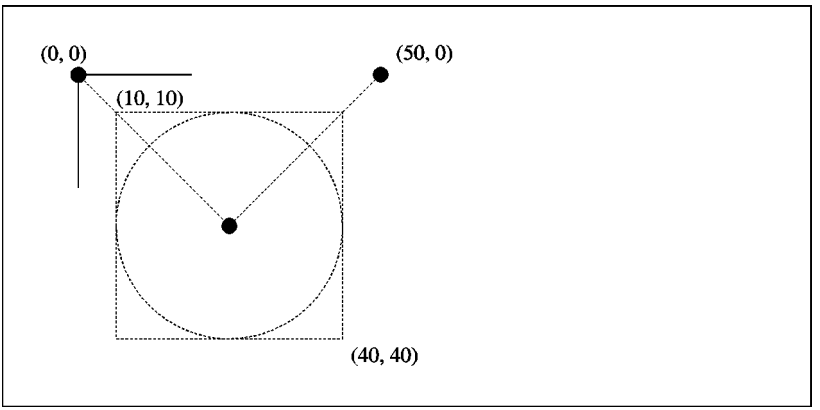


Abbildung 17.6 Das Ergebnis der Funktion Arc

Funktionen zum Zeichen von geometrischen Formen

Diese Funktionen verwenden den aktuellen Stift des angegebenen Bildschirmkontexts, um den Umfang zu zeichnen, und den aktuellen Pinsel, um das Innere der Form auszufüllen. Die aktuelle Position ist nicht betroffen.

Rectangle

Die Funktion *Rectangle* zeichnet ein Rechteck von der oberen linken Ecke zur unter rechten Ecke.

Die folgende Anweisung in einer *Paint*-Methode zeichnet ein Rechteck von (10,15) zu (100, 150):

```
Rectangle(PaintDC, 10, 15, 100, 150);
```

Abbildung 17.7 Das Ergebnis der Funktion Rectangle

RoundRect

Die Funktion *RoundRect* zeichnet ein Rechteck mit abgerundeten Ecken. Die abgerundeten Ecken sind als Viertel einer Ellipse definiert. Im folgenden Beispiel

zeichnet die Anweisung in einer *Paint*-Methode ein Rechteck von (10, 15) zu (100, 150), wobei die Ecken die Viertel einer Ellipse mit einer Breite von 9 und einer Höhe von 11 sind.

```
RoundRect(PaintDC, 10, 15, 100, 150, 9, 11);
```

Abbildung 17.8 Das Ergebnis der Funktion RoundRect

Ellipse

Die Funktion *Ellipse* zeichnet eine Ellipse, die von dem umgebenden Rechteck definiert wird.

Das folgende Beispiel zeichnet eine Ellipse innerhalb des Rechtecks (10, 50) zu (100, 150):

```
Ellipse(PaintDC, 10, 50, 100, 150);
```

Abbildung 17.9 Das Ergebnis der Funktion Ellipse

Pie und Chord

Die Funktionen *Pie* und *Chord* zeichnen Ellipsen-Segmente. Sie zeichnen beide Kreisbögen wie die Funktion *Arc*. *Pie* und *Chord* erstellen jedoch geometrische Formen.

Die Funktion *Pie* verbindet den Mittelpunkt der Ellipse mit den Endpunkten des Kreisbogens.

Die folgende *Pie*-Funktion zeichnet das obere Viertel eine Kreises innerhalb des Rechtecks (10, 10), (40, 40).

```
Pie(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

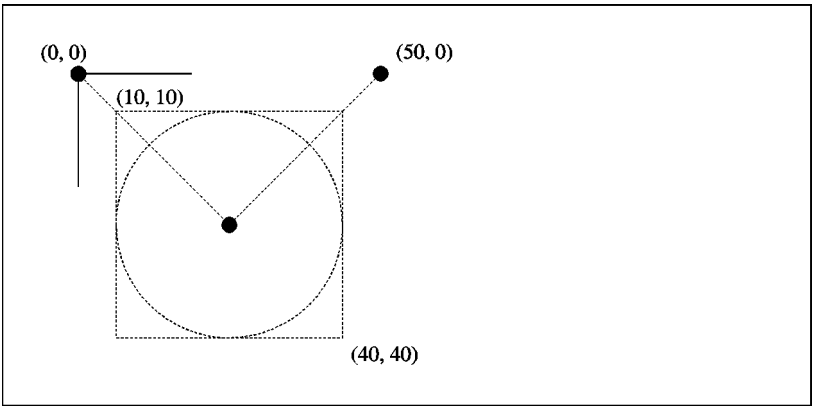


Abbildung 17.10 Das Ergebnis der Funktion Pie

Die Funktion *Chord* verbindet die beiden Eckpunkte des Kreisbogens:

```
Chord(PaintDC, 10, 10, 40, 40, 50, 0, 0, 0);
```

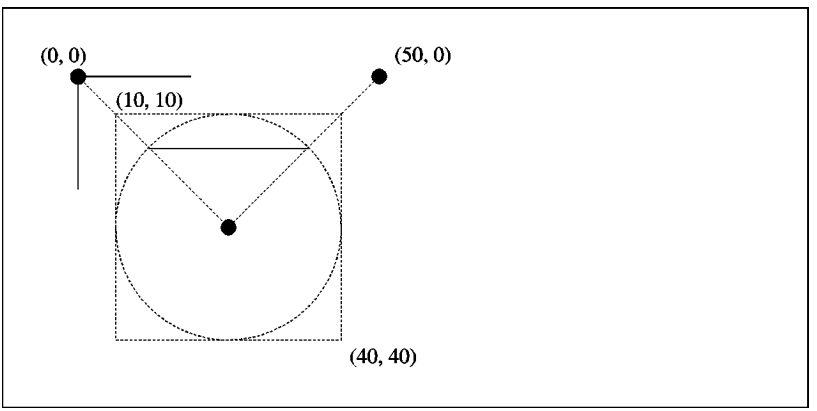


Abbildung 17.11 Das Ergebnis der Funktion Chord

Polygon

Die Funktion *Polygon* zeichnet zusammenhängende Liniensegmente, ähnlich wie die Funktion *Polyline*. Der Unterschied beruht darin, daß *Polygon* die Form schließt, indem ein Liniensegment vom angegebenen letzten zum angegebenen ersten Punkt gezogen wird. Schließlich wird das Polygon mit dem aktuellen Pinsel ausgefüllt, wobei der aktuelle Polygon-Füllmodus verwendet wird. Die folgende *Paint*-Methode zeichnet ein Dreieck.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo:
TPaintStruct);
var
  Points: array[0..2] of TPoint;
```

```
begin
  Points[0].X := 10;
  Points[0].Y := 15;
  Points[1].X := 10;
  Points[1].Y := 150;
  Points[2].X := 100;
  Points[2].Y := 150;
  Polygon(PaintDC, @Points, 3);
end;
```

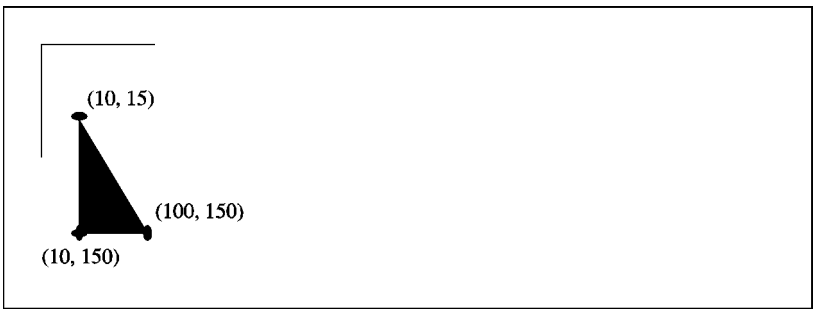


Abbildung 17.12 Das Ergebnis der Funktion Polygon

Der Einsatz von Paletten

Einige Ausgabegeräte können viele Farben anzeigen, aber nur einige davon zur gleichen Zeit. Die System- oder physikalische Palette enthält die Farben, die aktuell für die gleichzeitige Anzeige auf dem Ausgabegerät verfügbar sind. Windows gibt Ihrer Anwendung die teilweise Kontrolle über die Farben, die in die Systempalette des Ausgabegeräts übernommen werden. Wenn Ihre Anwendungen nur einfache Farben einsetzen, muß keine Palette direkt angewendet werden.

Die Änderung der Systempalette betrifft jedoch alle Zeichnungen auf dem Bildschirm und sogar andere Anwendungen. Mit anderen Worten: Eine Anwendung kann bewirken, daß alle anderen Anwendungen mit falschen Farben angezeigt werden. Der Windows-Paletten-Manager löst dieses Problem, indem er als Vermittler zwischen den Anwendungen fungiert, die die Systempalette ändern wollen. Windows weist jeder Anwendung eine virtuelle Palette zu, die die Gruppe der für die Anwendung erforderlichen Farben enthält. Der Paletten-Manager vergleicht die in der virtuellen Palette angeforderten Farben mit den in der Systempalette vorhandenen Farben. Ist die angeforderte Farbe nicht in der Systempalette vorhanden, fügt der Manager diese in die Systempalette ein. Wenn die virtuelle Palette mehr Farben festlegt, als die Systempalette aufnehmen kann, werden die zusätzlichen Farben der ähnlichsten Farbe in der Systempalette zugeordnet.

Wenn eine Anwendung aktiviert wird, kann sie die Systempalette mit Farben aus ihrer virtuellen Palette zu ergänzen. Diese Aktion hat unter Umständen die Folge, daß von anderen Anwendungen eingebrachte Farben »hinausgedrängt« werden. Generell reserviert Windows 20 permanente Farben in der Systempalette, die das Farbschema für alle Anwendungen und für Windows selbst aufrechterhalten.

Erstellen einer Farbpalette

Genauso wie Stifte und Pinsel (vgl. den Abschnitt »Zeichenwerkzeuge« weiter vorne in diesem Kapitel) sind auch virtuelle Paletten Zeichenwerkzeuge, die Sie mit der Funktion *CreatePalette* erzeugen können. Die Funktion benötigt einen Zeiger auf einen *TLogPalette*-Datenrecord als Parameter, erzeugt eine neue Palette und liefert ein Handle auf die Palette zurück, die an *SelectPalette* zur Auswahl für einen Bildschirmkontext übergeben werden kann. Der Record *TLogPalette* enthält Felder für die Windows-Versionsnummer (im Augenblick $0300), für die Anzahl der Paletteneinträge und für ein Array der Paletteneinträge. Jeder Paletteneintrag ist ein Record vom Typ *TPaletteEntry*. Der Typ *TPaletteEntry* hat drei Byte-Felder für die Farbfestlegung (*peRed*, *peGreen* und *peBlue*) und ein Feld für Flags (*peFlags*).

GetStockObject(Default_Palette) erzeugt eine voreingestellte Palette, die die 20 Farben enthält, die immer in der Systempalette vorhanden sind.

Wenn eine Anwendung ihre virtuelle Palette mit *SelectPalette* in den Bildschirmkontext eingebracht hat, muß diese vor Gebrauch realisiert werden. Dies führt die Windows-Funktion *RealizePalette* aus:

```
ThePalette := CreatePalette(@ALogPalette);
SelectPalette(TheDC, ThePalette, 0);
RealizePalette(TheDC);
```

RealizePalette bringt die Farben aus der virtuellen Palette der Anwendung in die Systempalette. Als erstes gleicht Windows die Farben ab, die sich bereits in der Systempalette befinden, und fügt dann die neuen Farben ein, solange genügend Platz dafür vorhanden ist. Schließlich werden die Farben, die nicht mit identischen Farben in der Systempalette gleichgesetzt werden können, an die ähnlichsten Farben in der Systempalette angeglichen.

Die Anwendung sollte ihre Palette realisieren, bevor mit dem Zeichnen begonnen wird – genauso wie die anderen Zeichenwerkzeuge vor dem Zeichnen ausgewählt werden müssen.

Zeichnen mit Paletten

Sobald die Anwendungspalette realisiert ist, kann das Zeichnen mit den Farben aus der Palette beginnen. Die Palettenfarben können direkt oder indirekt spezifiziert werden. Zur direkten Bestimmung einer Palettenfarbe verwenden

Sie den Paletten-Index *TColorRef*. *TColorRef* ist ein Longint-Wert, dessen höherwertiges Byte auf 1 gesetzt ist und dessen beide niederwertigen Bytes den Index eines Eintrags in der virtuellen Palette enthalten. So bestimmt $0100009 zum Beispiel den neunten Eintrag in einer Palette. Dieser Wert kann überall da eingesetzt werden, wo ein *TColorRef*-Argument erwartet wird, zum Beispiel:

```
ALogPen.lopnColor := $01000009;
```

Wenn Ihr Ausgabegerät eine volle 24-Bit-Farbdarstellung ohne Verwendung einer Systempalette zuläßt, beschränkt Sie ein Paletten-Index unnötigerweise auf die Farben in Ihrer virtuellen Palette. Diese Einschränkung können Sie umgehen, wenn Sie die Palettenfarben indirekt mit einem paletten-relativen *TColorRef*-Wert bestimmen. Ein paletten-relativer RGB-*TColorRef* ist derselbe Wert wie ein expliziter *TColorRef*-Wert, dessen höherwertiges Byte jedoch mit 2 besetzt ist. Die drei niederwertigen Bytes enthalten den RGB-Farbwert. So bestimmt zum Beispiel $020000FF einen paletten-relativen *TColorRef*-Wert für rot.

Wenn das Ausgabegerät eine Systempalette unterstützt, gleicht Windows die RGB-Information mit der ähnlichsten Farbe in der ausgewählten virtuellen Palette ab. Wird die Systempalette nicht unterstützt, dann wird *TColorRef* zur Festlegung eines expliziten RGB-Wertes verwendet.

Informationen über eine Palette

Windows definiert eine Funktion, mit deren Hilfe Sie sich über eine Palette informieren können. *GetPaletteEntries* hat einen Index, einen Bereich und einen Zeiger auf eine *TPaletteEntry*-Struktur als Argumente und füllt den Puffer mit den angegebenen Paletteneinträgen aus.

Ändern einer Palette

Die Einträge in einer virtuellen Palette können auf zwei Arten geändert werden. Die Funktion *SetPaletteEntries* hat dieselben Argumente wie *GetPaletteEntries* und ersetzt die angegebenen Einträge durch diejenigen, auf die das dritte Argument zeigt. Beachten Sie, daß Änderungen erst dann in der Systempalette abgeglichen werden, wenn *RealizePalette* aufgerufen wurde, und erst dann sichtbar werden, wenn der Client-Bereich erneut gezeichnet wird. Die Funktion *AnimatePalette* hat dieselben Argumente wie *SetPaletteEntries*, wird aber dann verwendet, wenn eine Anwendung die Palette schnell ändern und die Änderungen sofort sichtbar machen will. Wenn *AnimatePalette* aufgerufen wird, werden die Paletteneinträge, deren *peFlags*-Feld mit der Konstante *pc_Reserved* besetzt sind, durch die entsprechenden neuen Einträge ersetzt und sofort mit der Systempalette abgeglichen. Andere Einträge sind davon nicht betroffen.

Nehmen wir folgendes Beispiel an: Sie wollen die ersten zehn Einträge einer Palette ermitteln, ihre Werte durch Heraufsetzen des Rot-Anteils und Herab-

setzen des Blau- und Grün-Anteils ändern und die Änderungen sofort in Kraft treten lassen (vorausgesetzt, daß die Paletteneinträge mit *pc_Reserved* besetzt sind):

```
GetObject(ThePalette, SizeOf(NumEntries), @NumEntries);
if NumEntries >= 10 then
begin
  GetPaletteEntries(ThePalette, 0, 10, @PaletteEntries);
  for i := 0 to 9 do
  begin
    PaletteEntries[i].peRed := PaletteEntries[i].peRed + 40;
    PaletteEntries[i].peGreen := PaletteEntries[i].peGreen -
      40;
    PaletteEntries[i].peBlue := PaletteEntries[i].peBlue - 40;
  end;
  AnimatePalette(ThePalette, 0, 10, @PaletteEntries);
end;
```

Derselben Effekt könnte auch so erzielt werden:

```
SetPaletteEntries(ThePalette, 0, 10, @PaletteEntries);
RealizePalette(ThePalette);
```

Danach muß der Bildschirm neu gezeichnet werden, um die Farbänderungen anzuzeigen.

Beantworten von Palettenbotschaften

Wenn ein Fenster eine *wm_PaletteChanged*-Botschaft erhält, ist das ein Zeichen dafür, daß das aktive Fenster die Systempalette durch Realisieren seiner virtuellen Palette geändert hat. Das empfangende Fenster kann auf drei Arten antworten:

- Es kann nichts unternehmen (sehr schnelles Vorgehen, das aber zu unkorrekter Farbgebung führen kann);
- es kann seine virtuelle Palette realisieren und sich dann neu zeichnen (langsamer, aber die Farben sind dann so korrekt wie möglich);
- es kann seine Palette realisieren und dann mit der Funktion *UpdateColors* den Client-Bereich mit der Systempalette aktualisieren. *UpdateColors* ist schneller als das Neuzeichnen des Client-Bereichs, wobei aber die Farbgenauigkeit etwas leidet.

Das *WParam*-Feld des *TMessage* Records, das in einer *wm_PaletteChanged*-Botschaft übergeben wird, enthält das Handle auf das Fenster, das seine Palette realisiert. Wenn Sie als Antwort die Palette realisieren, müssen Sie zuerst sicherstellen, daß dieses Handle nicht das eigene Fenster-Handle ist. Sie würden ansonsten eine Endlosschleife erzeugen.

Das Programm PALTEST.PAS, das Sie auf Ihren Originaldisketten finden,

erstellt und realisiert eine virtuelle Palette mit acht Farben. Klicken Sie mit dem linken Mausschalter, wird ein Quadrat in jeder Farbe gezeichnet, klicken Sie mit dem rechten Mausschalter, werden die Farben in der Palette rotiert. Es wird ein *TColorRef*-Paletten-Index verwendet. Wird daher die virtuelle Palette geändert, ändern sich die Farben der Quadrate, wenn sie neu gezeichnet werden.

Die Funktion *PaletteIndex* ist für den Gebrauch von *TColorRef*-Paletten-Indizes hilfreich.

18. Details über Ressourcen

Windows-Programme sind aufgrund Ihrer standardisierten Oberfläche sehr einfach zu bedienen. So verwenden beispielsweise die meisten Windows-Anwendungen Menüs für die Befehlsauswahl und Zeigersymbole, mit denen der Mauszeiger die verschiedensten Werkzeuge (Pfeile, Pinsel, usw.) darstellen kann.

Menüs und Pfeile sind nur zwei Beispiele für die Ressourcen einer Windows-Anwendung. Ressourcen sind Daten, die in der .EXE-Datei der Anwendung gespeichert sind, sich aber nicht im normalen Datensegment des Programms befinden. Sie werden außerhalb des Quelltextes definiert und erst zum compilierten Code für die zu erzeugende .EXE-Datei hinzugebunden.

Die folgenden Ressourcen werden Sie am häufigsten erzeugen und verwenden:

- Menüs
- Dialogfenster
- Symbole
- Mauszeiger
- Accelerator-Tasten
- Bitmaps
- Strings

Beim Laden eines Programms in den Arbeitsspeicher des Computers beläßt Windows die Ressourcen normalerweise auf der Festplatte. Benötigte Ressourcen werden bei Bedarf erst während der Programmausführung einzeln geladen. Wird die Ressource nicht mehr benötigt, entfernt sie Windows aus dem Speicher (dies gilt für alle Ressourcen mit Ausnahme der Bitmaps).

Soll eine Ressource gleichzeitig mit dem Programm geladen werden oder soll Windows sie nicht aus dem Arbeitsspeicher entfernen, müssen ihre Attribute entsprechend geändert werden.

Die Erstellung und Modifizierung von Ressourcen wird im *Ressourcen-Editor* ausführlich beschrieben.

Erstellen von Ressourcen

Ressourcen können mit Hilfe eines Ressourcen-Editors oder eines Ressourcen-Compilers erstellt werden. Turbo Pascal für Windows unterstützt beide Methoden. Sie können also wählen, nach welcher Sie vorgehen wollen. In den meisten Fällen ist es einfacher, die Ressourcen mit Hilfe eines Editors auf dem Bildschirm zu erzeugen. Die Benutzung eines Ressourcen-Compilers bietet sich dagegen an, wenn Ressourcen-Makrodateien aus Büchern oder Zeitschriften compiliert werden sollen.

Gleichgültig, für welche Methode Sie sich entscheiden, für eine Anwendung wird normalerweise immer eine Ressourcen-Datei (.RES) erstellt. Diese Datei enthält Informationen in binärer Form für alle Menüs, Dialogfenster, Bitmaps und andere in der Anwendung eingesetzten Ressourcen.

Die binäre Ressourcen-Datei (.RES) wird während der Compilierung mit Hilfe der Compiler-Direktive *$R* zur ausführbaren Datei (.EXE) hinzugefügt. Zum Laden der Ressourcen in den Speicher muß entsprechender Code geschrieben werden. Um möglichst wenig Speicher zu belegen, wird jede Ressource separat geladen, sobald sie benötigt wird. Das Laden von Ressourcen wird im vorliegenden Kapitel ausführlich beschrieben.

Hinzufügen von Ressourcen zu einer .EXE-Datei

Nachdem die Ressourcen in Form einer .RES-Datei im binären Format vorliegen, müssen sie zur .EXE-Datei des Programms hinzugefügt werden. Dadurch entsteht eine Datei, die sowohl den compilierten Quelltext als auch die Ressourcen der Anwendung enthält.

Es gibt drei Möglichkeiten, Ressourcen zur .EXE-Datei hinzuzufügen:

1. Die Ressourcen werden mit dem Ressourcen-Editor von einer .RES-Datei in die bereits compilierte .EXE-Datei des Programms kopiert. Dieser Vorgang wird im *Ressourcen-Editor* ausführlich beschrieben.

2. In der Quelltextdatei wird eine Direktive spezifiziert. Das folgende Pascal-Programm fügt zum Beispiel die Ressourcen-Datei SAMPPROG.RES zur .EXE-Datei hinzu:

```
program SampleProgram;
{$R SAMPPROG.RES}
...
```

Zu jedem Pascal-Programm kann nur eine Ressourcen-Datei existieren (die allerdings weitere Ressourcen-Dateien enthalten kann). Es muß sich dabei immer um .RES-Dateien handeln, die die Ressourcen im binären Format enthalten. Mit der Compiler-Direktive *$R* kann eine einzelne .RES-Datei angegeben werden.

3. Es wird der Ressourcen-Compiler verwendet (er wird in diesem Kapitel ausführlich beschrieben).

Der Ressourcen-Compiler

Der Gebrauch der Ressourcen-Editoren wird im *Ressourcen-Editor* ausführlich erläutert. Der vorliegende Abschnitt enthält einen kurzen Überblick über die Erstellung von Ressourcen-Dateien mit dem Ressourcen-Compiler von Microsoft.

Der Ressourcen-Compiler ist eine DOS-Anwendung, die aus einer Ressourcen-Makrodatei eine binäre Ressourcendatei erzeugt. Es besteht auch die Möglichkeit, die Ressourcen mit dem Ressourcen-Compiler direkt zur .EXE-Datei hinzuzufügen. Die *$R*-Direktive von Turbo Pascal muß dann nicht benutzt werden. Dies ist vor allem dann hilfreich, wenn die Ressourcen eines Programms geändert werden müssen, ein Zugriff auf den Pascal-Quelltext des Programms aber nicht möglich ist.

Der Start des Ressourcen-Compilers kann nach dem DOS-Prompt oder in einem DOS-Fenster in Windows erfolgen. Das Format für den Start lautet:

```
RC [Optionen]  <.RC- oder .RES-Eingabedatei>  [.EXE-Ausgabedatei]
```

Die Einträge in eckigen Klammern sind optional. Sehen wir uns dazu ein Beispiel an:

```
RC -r SAMPPROG.RC
```

Dieser Befehl erzeugt aus einer Makrodatei SAMPPROG.RC die Datei SAMPPROG.RES. Die SAMPPROG.RES-Datei muß anschließend zur ausführbaren Datei SAMPPROG.EXE hinzugebunden werden. Dazu wird entweder die Compiler-Direktive *$R* verwendet, oder der Ressourcen-Compiler mit der .RES-Datei aufgerufen:

```
RC SAMPPROG.RES
```

Die binäre Ressourcen-Datei wird dadurch zur Datei SAMPPROG.EXE hinzugebunden, ohne daß der Turbo-Pascal-Compiler eingesetzt werden muß.

Es besteht auch die Möglichkeit, beide Schritte zu kombinieren, um eine vorhandene .EXE-Datei zu ändern und ihre Ressourcen zu aktualisieren. In diesem Fall muß der Befehl folgendermaßen lauten:

```
RC SAMPPROG.RC
```

Aus der Eingabe-Makrodatei SAMPPROG.RC wird so eine SAMPPROG.RES-Datei erzeugt und diese direkt zu SAMPPROG.EXE hinzugefügt. Die vorhandenen Ressourcen werden dadurch ersetzt.

In beiden Fällen muß das Pascal-Programm SAMPPROG.PAS Befehle zum Laden der Ressourcen in den Speicher enthalten.

Die folgende Tabelle listet die zulässigen Befehlszeilenoptionen für den Ressourcen-Compiler auf:

Tabelle 18.1 Befehlszeilenoptionen für den Ressourcen-Compiler

Option	Bedeutung
-r	Erzeugt eine .RES-Datei.
-l	Erzeugt eine Anwendung, die LIM 3.2 EMS verwendet.
-e	Erzeugt einen Treiber, der den EMS-Speicher nutzt.
-m	Setzt ein Flag für mehrere Instanzen.
-p	Erzeugt eine private Bibliothek.
-t	Erzeugt eine Anwendung, die nur im Protected Mode läuft.
-v	Protokoll (Meldungen über den Arbeitsfortschritt).
-d	Definiert ein Symbol.
-fo	Benennt eine .RES-Datei um.
-fe	Benennt eine .EXE-Datei um.
-i	Fügt einen Pfad für INCLUDE-Suchen hinzu.
-x	Ignoriert die Environment-Variable INCLUDE.
-k	Hält die Segmente in der .DEF-Dateireihenfolge bei (die Segmente werden nicht für schnelles Laden sortiert).

Makrodateien für den Ressourcen-Compiler

Bei einer .RC-Makrodatei handelt es sich um eine textuelle Definition der Ressourcen, die zu einem binären Format compiliert werden müssen. Die Festlegung von Ressourcen im Format des Ressourcen-Compilers entspricht in etwa dem Schreiben eines Programms. Wir wollen an dieser Stelle nicht die Makrosprache des Ressourcen-Compilers im Detail präsentieren, sondern anhand einiger Beispiele zeigen, wie Makrodateien compiliert und editiert werden können. Vergessen Sie nicht, daß die Erstellung komplexer Ressourcen in den meisten Fällen mit einem Ressourcen-Editor leichter zu bewerkstelligen ist. Auch Makrodateien können direkt mit dem Ressourcen-Editor erzeugt werden.

Das folgende Beispiel zeigt eine typische Ressourcen-Makrodatei:

```
; SAMPPROG.RC –a sample ressource script file
;             –this script illustrates the use of icons,
;               strings, menus, and dialog resources in Windows
;             –this script would be compiled with the resource
;               compiler and then appended to the pascal
;               program using the $R compiler directive

; include C-style header files for constant definitions:

#include <windows.h>
#include <wobjects.h>
#include <sampprog.h>

;define icons in separate files

SampleIcon     ICON    sample.ico

;strings can be defined numerically or symbolically
```

```
STRINGTABLE
BEGIN
  ids_NoProgram, "Program unavailable"
  ids_Invalid, "Invalid entry"
  ids_Fatal, "Fatal error!"
;  .
;  . other strings go here
;  .
END

; Menus define commands for the user

SampleMenu MENU
BEGIN
  POPUP "&File"
    BEGIN
      MENUITEM "&New..\tCtrl+N",       cm_FileNew
      MENUITEM "&Open..\tCtrl+O",      cm_FileOpen
      MENUITEM "&Save..\tCtrl+S",      cm_FileSave
      MENUITEM SEPARATOR
      MENUITEM "&Print \tCtrl+P",      cm_FilePrint
    END
  POPUP "&Edit"
    BEGIN
    MENUITEM "&Undo \tAlt+BkSp",       cm_EditUndo
    MENUITEM SEPARATOR
    MENUITEM "&Cut \Shift+Del",        cm_EditCut
    MENUITEM "&Copy \tIns",            cm_EditCopy
    MENUITEM "&Paste \tShift+Ins",     cm_EditPaste
    MENUITEM "&Delete \tDel",          cm_EditDelete
    MENUITEM "&Clear All \tCtrl+Del",  cm_EditClear
    END
  POPUP "&View"
    BEGIN
      MENUITEM "Summary \tF2",         cm_ViewSummary
      MENUITEM "Graph \tF3",           cm_ViewGraph
    END
END

; Accelerators provide "hot keys" to menus, commands

SampleAccelerators ACCELERATORS
BEGIN
  "^n", cm_FileNew
  "^o", cm_FileOpen
  "^s", cm_FileSave
  "^p", cm_FilePrint
  VK_DELETE, cm_EditCut, VIRTKEY, SHIFT
  VK_INSERT, cm_EditCopy, VIRTKEY,
  VK_INSERT, cm_EditPaste, VIRTKEY, SHIFT
```

```
  VK_DELETE, cm_EditDelete, VIRTKEY,
  VK_DELETE, cm_EditClear, VIRTKEY, CONTROL
  VK_BACK, cm_EditUndo, VIRTKEY, ALT
  VK_F2, cm_ViewSummary, VIRTKEY,
  VK_F3, cm_ViewGraph, VIRTKEY,
END

; Dialog boxes display information to the user

AboutBox DIALOG 20, 20, 160, 80
CAPTION "About SAMPPROG"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  CTEXT "Sample Program"              -1,   0,  10,  160,  10
  CTEXT "Written in Turbo Pascal"     -1,   0,  26,  160,  10
  CTEXT "for Windows 3.0"             -1,   0,  36,  160,  10
  ICON "Sample_Icon"                  -1,   8,   8,    0,   0
  DEFPUSHBUTTON "OK",                IDOK,  60,  56,   40,  14
END

FileOpen DIALOG 20, 20, 204, 124
CAPTION "File Open"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
FONT 8, "Helv"
BEGIN
  LTEXT    "File&name:",        -1,   6,   8,  36,  10
  EDITTEXT                     100,  42,   6,  98,  12, ES_AUTOSCROLL
  LTEXT    "Directory:",        -1,   6,  20,  36,  10
  LTEXT    ""                  101,  42,  20,  98,  10
  LTEXT    "&Files:",           -1,   6,  32,  64,  10
  LISTBOX  102, 6, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  LTEXT    "&Directories:",     -1,  76,  32,  64,  10
  LISTBOX  103, 76, 44, 64, 82, WS_TABSTOP | WS_VSCROLL | LBS_SORT
  DEFPUSHBUTTON "OK",         IDOK, 146,  6,  50,  14
  PUSHBUTTON    "Cancel", IDCANCEL, 146, 24,  50,  14
END
```

Die Header-Datei SAMPPROG.H enthält Konstantendefinitionen im C-Stil. Eine typische Header-Datei sieht folgendermaßen aus:

```
/* SAMPPROG.H    —header file of constants for SAMPPROG.RC /*

#define ids_NoProgrm       601
#define ids_Invalid        602
#define ids_Fatal          603

#define cm_FileNew         701
#define cm_FileSave        702
/*
```

```
    .
    . other constants included here
    .
*/
```

Hinweise für die Benutzung des Ressourcen-Compilers

Bei der Arbeit mit der Makrosprache des Ressourcen-Compilers ist folgendes zu beachten:

1. Kommentare beginnen mit einem Strichpunkt (;) und enden am Zeilenende. Es können auch Kommentare im C-Stil verwendet werden (Text zwischen /* und */).

2. Der Ressourcen-Compiler unterscheidet bei Schlüsselwörtern wie BEGIN und END *nicht* zwischen Groß- und Kleinschreibung.

3. Bei Symbolnamen wie *cmFileNew* und *AboutBox* wird zwischen Groß- und Kleinschreibung unterschieden.

4. Der Ressourcen-Compiler berücksichtigt keine Whitespaces.

5. Zur Aufnahme von Header-Dateien mit Konstanten muß die #include-Direktive verwendet werden. Diese Konstanten werden mit einem #define-Befehl im C-Stil definiert.

6. Zur Aufnahme anderer Ressourcen-Makrodateien (*.RC oder *.DLG) kann die #rcinclude-Direktive verwendet werden. Auf diese Weise können zum Beispiel compilierte Dialogfenster einbezogen werden.

7. Anstelle von Zahlen sollten für Konstanten wie String-IDs, Menübefehl-IDs, etc. symbolische Konstantennamen verwendet werden. Ihre Definition erfolgt normalerweise in einer Header-Datei mit Hilfe von Konstantendefinitionen im C-Stil. Zur symbolischen Referenz auf diese Konstanten müssen außerdem entsprechende Pascal-Konstanten definiert werden.

8. Jede Konstante für eine Menüfeld-ID bzw. String-ID muß eindeutig sein.

9. Zur Erzeugung einer hierarchischen Struktur können Menüs beliebig verschachtelt werden.

10. In Menüpunkten und Steuerelementen von Dialogen kann ein Zeichen durch Voranstellen des et-Zeichens (&) unterstrichen werden. Der betreffende Menüpunkt bzw. das Steuerelement kann dann durch gleichzeitiges Drücken der Taste Alt und des unterstrichenen Zeichens aufgerufen werden.

11. In Menüpunkten kann zur Ausrichtung von Text mit \t ein Tabulatorzeichen erzeugt werden. Auf diese Weise können zum Beispiel Tastaturbefehl-Äquivalente ausgerichtet werden.

12. Tastenkürzel müssen in einer Accelerator-Tabelle definiert werden, da der in Menüs enthaltene Text die Accelerator-Tasten nicht automatisch erzeugt.

13. Steuerelemente, auf die das Programm nicht zugreift (zum Beispiel statischer Text), haben normalerweise die ID -1.

14. Die Position und Größe von Dialogfenstern und Steuerelementen wird in der Reihenfolge Links, Oben, Breite, Höhe angegeben.

15. Menüpunkte, Dialogfenster und Steuerelemente wie Bildlaufleisten, Editierfelder und Schalter können mit Windows-Stil-Konstanten modifiziert werden. Diese Konstanten werden mit dem bitweisen or-Operator im C-Stil, einem senkrechten Strich (|) hinzugefügt.

Steuerelemente können beispielsweise die Stile *ws_Child*, *ws_tabStop* und *ws_Border* haben.

16. Ressourcen-Makrodateien müssen mit dem Ressourcen-Compiler compiliert werden.

17. Meist ist es einfacher, einen Ressourcen-Editor zu verwenden, anstatt komplexe Ressourcen-Definitionen mit Makrodateien zu schreiben.

18. Im Turbo-Pascal-Programm muß Code enthalten sein, der das Laden und die Verwendung der Ressourcen veranlaßt.

Laden von Ressourcen in einer Anwendung

Nachdem eine Ressource zur ausführbaren Datei hinzugefügt wurde, muß sie von der Anwendung explizit geladen werden. Die Vorgehensweise beim Laden hängt von der Art der jeweiligen Ressource ab.

Laden von Menüs

Das Menü eines Fensters ist ein Erstellungsattribut des Fensters, d.h. es muß vor dem entsprechenden Fensterelement mit einer *Create*-Methode erzeugt werden. Ein Menü muß daher im *Init*-Konstruktor des Fenstertyps oder kurz nach der Konstruktion angegeben werden. Das Laden einer Menü-Ressource erfolgt bei der Konstruktion eines neuen Fensterobjekts durch einen Aufruf der Windows-Funktion *LoadMenu*. Dabei muß der ID-String für das Menü angegeben werden:

```
constructor SampleMainWindow.Init(AParent: PWindowsObject)
  ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ...
end;
```

PChar(100) wandelt die Integerzahl 100 in den Typ PChar um. PChar ist ein zu Windows kompatibler Stringtyp.

LoadMenu lädt die Menü-Ressource mit einer ID von 100 in das neue Fensterobjekt. Eine Ressource muß nicht unbedingt eine numerische ID haben, sie kann mit einem symbolischen Namen (einen String) bezeichnet werden, zum Beispiel 'SampleMenu'. In diesem Fall müßte der obige Code folgendermaßen geschrieben werden:

```
constructor SampleMainWindow.Init(AParent: PWindowsObject)
  ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'SampleMenu');
  ...
end;
```

Das Erstellen von Fensterobjekten wird in Kapitel 10 »Fensterobjekte« ausführlich beschrieben.

Zur Bearbeitung einer Menüauswahl wird eine Methode für das Fenster definiert, zu dem das Menü gehört. Dazu wird der Kopf der Methodendefinition mit dem *cm_First*-Bezeichner erweitert:

```
procedure HandleMenu101(var Msg: TMessage); virtual cm_First + 101;
```

101 ist die ID der gewählten Menüauswahl (nicht diejenige der Menü-Ressource). Jede Menüauswahl hat eine eindeutige, aus einem Integer bestehende ID. In dieser Methode müssen Aufgaben definiert werden, die auf die Menüauswahl korrekt antworten.

Laden von Accelerator-Tabellen

Accelerator-Tasten sind Tastenkürzel für den schnellen Aufruf von Befehlen. Sie werden meist als als Abkürzung für Menübefehle definiert. Die Del-Taste wird zum Beispiel häufig als Alternative zur Auswahl des Löschen-Befehls in einem Editiermenü verwendet.

Accelerator-Tasten können aber auch Befehle implementieren, die nicht in einem Menü enthalten sind.

Accelerator-Ressourcen werden in einer Accelerator-Tabelle gespeichert. Das Laden einer Accelerator-Tabelle erfolgt durch einen Aufruf der Windows-Funktion *LoadAccelerators*, die ein Handle auf die Tabelle zurückliefert. Im Gegensatz zu einer Menü-Ressource, die mit einem bestimmten Fenster verknüpft ist, gehört die Accelerator-Ressource zur gesamten Anwendung. Jede Anwendung kann nur eine Accelerator-Tabelle haben.

Anwendungsobjekte reservieren zum Speichern des Handle auf die Accelerator-Ressource ein Objektfeld namens *HAccTable*. Normalerweise erfolgt das Laden der Accelerator-Ressource in der *InitInstance*-Methode des Anwendungsobjekts:

```
procedure SampleApplication.InitInstance;
begin
  TApplication.InitInstance;
  HAccTable := LoadAccelerators(HInstance, 'SampleAccelerator');
end;
```

Meist werden Sie die Accelerator-Tasten als Kürzel für Menüauswahlen definieren. Zum Beispiel wird [Shift][Ins] häufig als Kürzel für den Menüpunkt *Einfügen* verwendet. Die Eingabe einer Accelerator-Taste erzeugt eine befehlsbezogene Botschaft identisch mit der, die durch die Auswahl eines Menüpunkts ausgelöst wird. Um die Antwortmethode für eine Menüauswahl mit einem Accelerator zu verknüpfen, muß der Wert des Accelerators in der Ressource mit der ID der Menüauswahl übereinstimmen.

Laden von Dialogfenstern

Dialogfenster haben als einziger Ressourcen-Typ korrespondierende Objekttypen in ObjectWindows. *TDialog* und seine Nachkommen (einschließlich *TDlgWindow*) definieren Schnittstellenobjekte, die Dialogfenster-Ressourcen verwenden. Jedes Dialogfensterobjekt ist normalerweise mit einer Dialogfenster-Ressource verknüpft, die seine Größe, seine Position und die Anordnung seiner Steuerelemente (beispielsweise Schalter und Listen) festlegt.

Die Ressource für ein Dialogfensterobjekt wird bei der Konstruktion des Dialogfensters angegeben. Dialogfenster-Ressourcen können – ebenso wie Menü- oder Accelerator-Ressourcen – einen symbolischen Namen oder eine Integer-ID haben:

```
ADLg := New(PSampleDialog, Init(@Self, 'AboutBox1'));
```

oder

```
ADLg := New(PSampleDialog, Init(@Self, PChar(PChar(120))));
```

Die Erstellung von Dialogfenstern wir in Kapitel 5 »Konstruktion eines Dialogs« ausführlich erläutert.

Laden von Mauszeigern und Symbolen

Zu jedem Fensterobjekttyp gehören spezielle Attribute, die sogenannten *Registrierungsattribute*. Dazu gehören zum Beispiel die Mauszeiger und Symbole des Fensters. Zum Setzen dieser Attribute definieren Sie eine Methode mit dem Namen *GetWindowClass* (und eine namens *GetClassName*).

Angenommen, Sie haben ein Zeigersymbol zur Auswahl von Einträgen in einer Liste erstellt. Das Zeigersymbol hat die Form eines Zeigefingers und ist in einer Mauszeiger-Ressource mit dem Namen »Finger« gespeichert. Außerdem haben Sie ein Symbol namens »SampleIcon« erzeugt, das ein Gesicht darstellt.

Zum Laden dieser Ressourcen schreiben Sie die folgende *GetWindowClass*-Methode:

```
procedure SampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  TWindow.GetWindowClass(AWndClass);
  AWndClass.hCursor := LoadCursor(HInstance, 'Finger');
  AWndClass.hIcon := LoadIcon(HInstance, 'SampleIcon');
end;
```

Zwischen Mauszeigern und Symbolen existiert folgender Unterschied: Während ein Mauszeiger nur für ein bestimmtes Fenster definiert wird, repräsentiert ein Symbol die gesamte Anwendung. Das Symbol wird deshalb nur im Objekttyp des Hauptfensters eingerichtet.

Von dieser Regel gibt es allerdings ein Ausnahme: Folgt die Anwendung dem Standard für die Multidokumentschnittstelle (MDI), hat jedes untergeordnete MDI-Fenster sein eigenes Symbol. Wenn Sie die vordefinierten Mauszeiger und Symbole von Windows verwenden wollen, müssen Sie anstelle von *HInstance* eine 0 übergeben und einen *idc_* Wert (z.B. *idc_IBeam*) für einen Mauszeiger bzw. einen *idi_* Wert (z.B. *idi_Hand*) für ein Symbol verwenden. Dazu ein Beispiel:

```
procedure SampleWindow.GetWindowClass(var AWndClass: TWndClass);
begin
  TWindow.GetWindowClass(AWndClass);
  AWndClass.hCursor := LoadCursor(0, idc_IBeam);
  AWndClass.hIcon := LoadIcon(0, idi_Hand);
end;
```

Auf Registrierungsattribute für Fenster wird in Kapitel 10 »Fensterobjekte« ausführlich eingegangen.

Laden von String-Ressourcen

Die Strings einer Anwendung werden als Ressourcen definiert, um die Anpassung der Anwendung an bestimmte Gegebenheiten zu vereinfachen oder ihre Übersetzung in andere Sprachen zu erleichtern. Wenn die Strings im Quelltext enthalten sind, muß für jede Änderung oder Übersetzung auf den Quelltext zugegriffen werden. Sind sie dagegen als Ressourcen definiert, werden sie in einer Stringtabelle innerhalb der .EXE-Datei einer Anwendung gespeichert. Die Strings können dann mit einem String-Editor übersetzt und geändert werden, ohne daß auf den Quelltext zugegriffen werden muß. Zu jeder .EXE-Datei kann jeweils nur eine Stringtabelle existieren. Mit der Funktion *LoadString* kann ein String aus der Stringtabelle in einen Puffer des Datensegments der Anwendung geladen werden. Die Syntax für *LoadString* lautet:

```
LoadString(HInstance, StringID, @TextItem, SizeOf(TextItem));
```

- Der Parameter *StringID* ist die ID des Strings (z.B. 601) in der Stringtabelle. Anstelle der Nummer kann auch eine Konstante verwendet werden.

- Der Parameter *@TextItem* ist ein Zeiger auf ein Zeichenarray (PChar), das den String erhält.

- Der Parameter *SizeOf(TextItem)* gibt die maximale Anzahl der Zeichen an, die in *@TextItem* kopiert werden können. Die maximale Größe einer String-Ressource liegt bei 255 Zeichen. Wenn Sie sicherstellen wollen, daß der gesamte String eingetragen wird, geben Sie 256 für die Größe des Zeichenpuffers an.

LoadString liefert die Anzahl der Zeichen zurück, die in den Puffer kopiert wurden. Existiert die Ressource nicht, liefert *LoadString* 0 zurück. Mit Hilfe einer String-Ressource können Sie Text in einem Meldungsfenster anzeigen. Angenommen, es soll die Fehlermeldung Program unavailable angezeigt werden. Dazu wird der entsprechende String in einer Stringtabelle definiert und seine ID als Konstante *ids_NoPrgrm* festgelegt. Damit diese Meldung als String-Ressource verwendet werden kann, wird folgende Prozedur geschrieben:

```
procedure TestDialog.RunErrorBox(ErrorNumber: Integer); virtual;
var
  TextItem: array[0..255] of Char;
begin
  LoadString(HInstance, ids_NoPrgrm, @TextItem, 20);
  MessageBox(HWindow, @TextItem, 'Error', mb_OK or
    mb_IconExclamation);
end;
```

Diese Prozedur lädt einen einzelnen String in ein Fehlermeldungsfenster. Soll eine Stringliste in eine Liste geladen werden, muß jeder String einzeln mit *LoadString* geladen und dann mit *AddString* zur Liste hinzugefügt werden.

Mit Hilfe von String-Ressourcen können auch Menüauswahlen an vorhandene Menüs angehängt werden, ohne auf den Quelltext zuzugreifen. Dazu wird zunächst der String mit *LoadString* geladen und anschließend in Aufrufen der Windows-Funktionen *CreateMenu* und *AppendMenu* als Parameter übergeben:

```
constructor SampleWindow.Init(AParent: PWindowsObject; ATitle:
PChar);
var
  TextItem array[0..255] of Char;
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  LoadString(HInstance, 301, @TextItem, 20);
  AppendMenu(Attr.Menu, mf_String or mf_Enabled, 501, @TextItem);
end;
```

Laden von Bitmaps

Bitmap-Ressourcen werden mit der Windows-Funktion *LoadBitmap* in den Speicher geladen. Die Funktion liefert ein Handle auf das Bitmap zurück.

Zum Beispiel lädt

```
HMyBit := LoadBitmap(HInstance, PChar(501));
```

die Bitmap-Ressource mit der ID 501 und speichert das zurückgelieferte Bitmap-Handle in der Variable *HMyBit*.

Ist ein Bitmap einmal geladen, bleibt es im Speicher, bis es explizit gelöscht wird. Dies ist – im Gegensatz zu anderen Ressourcen – auch dann der Fall, wenn der Anwender das Programm beendet.

Windows verfügt über eine Anzahl vordefinierter Bitmaps, die Bestandteil der grafischen Schnittstelle von Windows sind. Diese Bitmaps, zum Beispiel *obm_DnArrow*, *obm_Close* oder *obm_Zoom*, können von Ihrer Anwendung geladen werden. Zum Laden vordefinierter Bitmaps könnenSie – wie bei vordefinierten Symbolen und Mauszeigern – im *LoadBitmap*-Aufruf 0 für den Parameter *HInstance* einsetzen:

```
HMyBit := LoadBitmap(0, PChar(obm_Close));
```

Nachdem das Bitmap geladen wurde, kann es in der Anwendung auf verschiedene Weise eingesetzt werden:

- Zum Zeichnen eines Bildes auf dem Bildschirm. Beispielsweise kann zum Zeichnen eines Logos ein Bitmap in das *About*-Fenster einer Anwendung geladen werden.
- Zum Erstellen eines Pinsels. Mit dem Pinsel kann dann ein Bereich des Bildschirms gefüllt oder der Hintergrund eines Fensters (zum Beispiel ein Streifenmuster) erzeugt werden.
- Zum Anzeigen von Bildern anstelle von Text in Menüs oder Listen. Beispielsweise kann anstelle des Worts »Pfeil« für eine Menüauswahl das Bild eines Pfeiles angezeigt werden.

In Kapitel 17 »Die graphische Geräteschnittstelle« wird die Verwendung von Bitmaps ausführlich erläutert.

Wenn das Bitmap nicht mehr benötigt wird, sollte es aus dem Speicher gelöscht werden, da der belegte Speicherplatz sonst nicht von anderen Anwendungen genutzt werden kann. Löschen Sie das Bitmap spätestens vor der Beendigung der Anwendung. Zum Löschen wird die Windows-Funktion *DeleteObject* verwendet:

```
if DeleteObject(HMyBit) then { erfollgreich };
```

Nach dem Löschen des Bitmaps ist das Handle ungültig und kann nicht mehr verwendet werden.

Erstellung eines Pinsels mit Hilfe eines Bitmaps

Zum Füllen einer Fläche auf dem Bildschirm können Sie mit Hilfe eines Bitmaps einen Pinsel erstellen. Die Fläche kann einfarbig oder mit einem Muster gefüllt werden.

Ein Bitmap, das als Pinsel benutzt werden soll, muß mindestens eine Fläche von 8 mal 8 Pixeln abdecken. Ist das Bitmap größer, wird nur eine 8 mal 8 Pixel große Fläche in seiner linken oberen Ecke als Pinsel verwendet.

Angenommen, Sie wollen eine Fläche mit einem Streifenmuster füllen, wie in Abbildung 18.1 gezeigt. Dazu erstellen Sie mit einem Ressourcen-Editor das zweistreifige Muster (vgl. Abbildung 18.2). Es muß mindestens 8 mal 8 Pixel groß sein.

Anschließend können Sie das Bitmap in Ihrem Quelltext verwenden und damit einen Pinsel erstellen. Zum Füllen der gesamten Fläche (wie in Abbildung 18.1 gezeigt) kopiert Windows den Pinsel mehrmals.

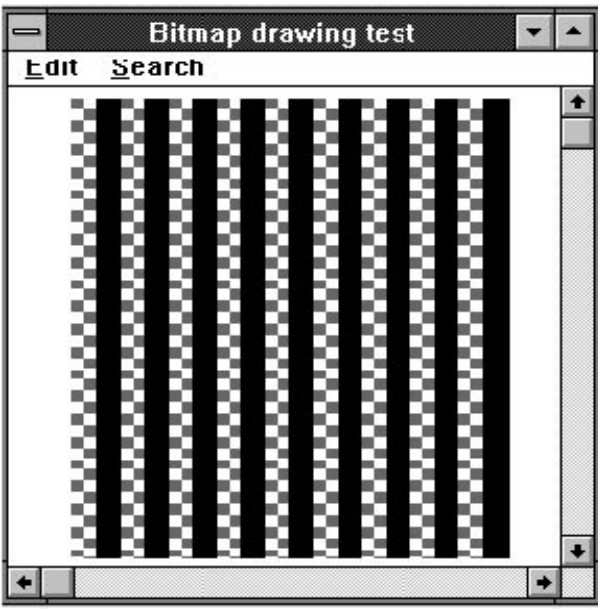


Abbildung 18.1 Streifenmuster, das einen Bereich des Bildschirms füllt

Das eigentliche Bitmap hat nur eine Größe von 8 mal 8 Pixeln, mit Hilfe des Pinsels kann aber der gesamte Bildschirm gefüllt werden.

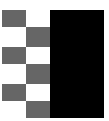


Abbildung 18.2 Bitmap-Ressource zur Erzeugung eines Pinsels für das Muster in Abbildung 18.1

Das folgende Programm legt das Bitmap-Muster für den Pinsel fest:

```
procedure SampleWindow.MakeBrush;
var
  MyLogBrush: TLogBrush;
begin
  HMyBit := LoadBitmap(HInstance, PChar(502));
  MyLogBrush.lbStyle := bs_Pattern;
```

```
  MyLogBrush.lbHatch := HMyBit;
  TheBrush := CreateBrushIndirect(@MyLogBrush);
end;
```

Zum Testen des Musters wird es in einem Rechteck angezeigt:

```
procedure MyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
begin
  SelectObject(PaintDC, TheBrush);
  Rectangle(PaintDC, 20, 20, 200, 200);
end;
```

Bitmap und Pinsel müssen nach Gebrauch des Pinsels gelöscht werden:

```
DeleteObject(HMyBit);
DeleteObject(TheBrush);
```

Anzeigen von Bitmaps in Menüs

Die Anzeige eines Bitmaps in einem Menü erfolgt mit der Funktion *ModifyMenu*. Ein vorhandener Menüpunkt wird dabei so verändert, daß anstelle des Menütexts, der für den Menüpunkt in einem Menüeditor definiert wurde, das Bitmap angezeigt wird. Im folgenden Beispiel fügt der *Init*-Konstruktor ein Menü zu einem Fenster hinzu und ändert es:

```
type
  MyLong = record
    case Integer of
      0: (TheLong: Longint);
      1: (Lo: Word;
          Hi: Word);
  end;

constructor SampleWindow.Init(AParent: PWindowsObject; ATitle:
  PChar);
var
  ALong: MyLong;
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, PChar(100));
  ALong.Lo := LoadBitmap(HInstance, PChar(503));
  ModifyMenu(Attr.Menu, 111, mf_ByCommand or mf_Bitmap, 211,
    PChar(ALong.TheLong));
  ...
end;
```

In diesem Beispiel ist 111 die Befehls-ID der Menüauswahl, die geändert werden soll, 211 ist die neue ID. Es könnte aber auch dieselbe ID verwendet werden.

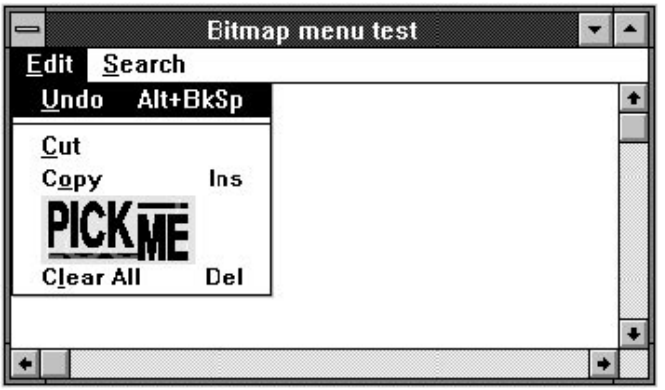


Abbildung 18.3 Ein Menü mit einem Bitmap

19. Standardrichtlinien für Anwendungen

Für die Entwicklung von Windows-Anwendungen sprechen vor allem die leichte Erlernbarkeit und der einfache Gebrauch dieser Programme. Zu der standardisierten Oberfläche der Anwendungen gehört es, daß Operationen, die allen Programmen gemeinsam sind, auf dieselbe Weise angeboten werden. So hat zum Beispiel das Edit-Menü sowohl in Text- als auch in Grafik- und Zeichenanwendungen immer dieselben Menüpunkte und Tastenkürzel. Mit anderen Worten: Wenn Sie mit einer Windows-Anwendung umgehen können, können Sie mit allen anderen auch umgehen.

Um die Richtlinien für den Zugriff auf gemeinsame Operationen zu formalisieren, hat IBM Regeln aufgestellt, die unter dem Begriff *Common User Access* (CUA) zusammengefaßt sind. Obwohl diese Regeln für OS/2 Presentation-Manager-Anwendungen aufgestellt wurden, haben sie auch für Windows-Anwendungen Gültigkeit und werden generell als Industriestandard angesehen. Dieses Kapitel beruht auf den von IBM veröffentlichten CUA-Richtlinien.

Entwurfsprinzipien

Für Anwendungen, die sich an die CUA-Richtlinien halten, gibt es zwei vornehmliche Ziele: Der Anwender soll die Wirkung jeder Anwendungsaktion sicher einschätzen und den Fluß der Ereignisse steuern können.

Antworten, die der Erwartung des Anwenders entsprechen

Wenn eine Anwendung konsistente und der aktuellen Situation entsprechende Antworten gibt, kann der Anwender leichter ein gewißes Vertrauen zu ihr entwicklen. Überdenken Sie in dieser Hinsicht die folgenden Vorschläge:

Stellen Sie Analogien her. Gestalten Sie das Verhalten und die Eigenschaften einer Anwendung so, daß sie auf Grundlagen basiert, die dem Anwender vertraut und bekannt sind. Entwerfen Sie Ihre Anwendung also analog zu einem bereits bestehenden, nicht »computerisierten« Verfahren. Der Anwender wird so sehr

viel schneller und leichter das der Anwendung zugrundeliegende Konzept erfassen und akzeptieren – und er wird sich schnell mit der Anwendung vertraut fühlen.

Achten Sie auf Konsistenz. Achten Sie auf die Konsistenz der Anwendung in Bezug auf Erscheinungsbild und Gebrauch. Verwenden Sie gleiche Befehle (Auswahl aus Menüs, Anklicken von Symbolen, etc.) für gleiche Operationen. Gestalten Sie Bildschirme für ähnliche Aufgaben auch in ähnlicher Weise.

Vermeiden Sie Modi. Der Modus ist ein Zustand der Anwendung, der den Anwender-Zugriff auf bestimmte Eigenschaften beschränkt. Modi werden oft dazu mißbraucht, um die Aktivitäten des Anwenders in eine ganz bestimmte Richtung zu lenken. Modi sind bei kurzen Fehlermeldungen und bei der Verwendung von Werkzeugen durchaus angebracht. Wenn der Mauszeiger zum Beispiel ein Stift ist, kann der Anwender zeichnen, aber nicht Formen verschieben. Sorgen Sie für eine optische Darstellung der Modi, zum Beispiel über wechselnde Darstellungen des Mauszeigers oder grau hinterlegte Menüauswahlpunkte.

Anwender-Steuerung des Programmflusses

Der Anwender fühlt sich sicherer, wenn er den Ablauf der Aktivitäten steuern kann. Die Kontrolle über die Arbeitsschritte läßt kein Gefühl von »Geheimnis« entstehen, das ja Computerprogramme immer noch umgeben soll. Das Programm soll schließlich zu einem wirkungsvollen Mittel zum Zweck werden.

Listen Sie zur Auswahl stehende Möglichkeiten auf. Stellen Sie dem Anwender eine Liste zur Verfügung, die die in Frage kommenden Aktionen enthält. Normalerweise beinhalten Methoden eine Palette von Werkzeugen oder Farben und ein Menü mit Optionen.

Stellen Sie Rückmeldungen her. Jede Anwender-Aktion sollte eine optische oder akustische Rückmeldung zur Folge haben.

Unterstützen Sie das Experimentieren mit der Anwendung. Hindern Sie Anwender nicht am Ausprobieren von Möglichkeiten. Das Experimentieren ist häufig der beste Weg zum Kennenlernen einer neuen Anwendung. Bauen Sie unbedingt eine Widerrufen-Option in die Anwendung ein.

Standard für Erscheinungsbild und Verhalten

Der Programm-Manager und die Beispielprogramme zeigen bereits, wie der Standard für Erscheinungsbild und Verhalten einer Anwendung aussehen sollte. So wird jede Anwendung z.B. in einem Hauptfenster angezeigt oder einfacher und doppelter Mausklick haben in jeder Anwendung ähnliche Resultate.

Generelle Präsentation

Halten Sie sich an die Abfolge Objekt – Aktion. Folgende Abfolge von Ereignissen ist typisch für eine Anwendungsoperation: Der Anwender wählt einen Eintrag oder ein Objekt aus und führt eine Aktion damit aus. So kann der Anwender zum Beispiel in einem Textprogramm Text auswählen oder markieren und dann die Schrift oder die Größe ändern. In einem Zeichenprogramm wählt der Anwender zum Beispiel ein Rechteck aus und verschiebt es an eine andere Position auf dem Bildschirm. Obwohl es eine Vielzahl von Möglichkeiten für die Auswahl von Objekten gibt und womöglich eine noch größere Anzahl von verschiedenen Möglichkeiten für die Auswahl von Aktionen, sollten Sie dennoch versuchen, sich an dieses Grundmodell zu halten.

Wählen Sie Ihre Fenster mit Bedacht. Eine Windows-Anwendung kann in einem oder mehreren Fenstern angezeigt werden. Damit Ihre Anwendung von optimalem Nutzen ist, müssen Sie sorgfältig mit den verschiedenen Fenstertypen und ihren Einsatzmöglichkeiten umgehen. Das *Hauptfenster* sollte als Basis für die Anwendung dienen, sozusagen ihr »Rückgrat« sein. Es repräsentiert die gesamte Anwendung dadurch, daß seine Minimierung zu einem Symbol sämtliche Aktionen des Programms suspendiert. Der Anwender kann das Hauptfenster aber auch maximieren, so daß es den gesamten Bildschirm einnimmt. *Sekundäre* Fenster, wie Pop-up- oder Dialogfenster, sollten vom Hauptfenster als Ergebnis einer Hauptfenster-Aktion eingeblendet werden. Pop-up-Fenster können ein separates Modul der Anwendung repräsentieren, sollten aber nicht minimiert werden.

Maus- und Tastatur-Interaktionen

Zeigen Sie die Auswahl eines Objekts an. Wenn ein Objekt durch einen Mausklick ausgewählt wurde, sollten Sie dies optisch darstellen, indem Rechtecke an den Ecken des gewählten Objekts eingeblendet oder Farbe und/oder Dicke geändert werden. Diese Auswahlmarkierung darf aber nicht dauerhaft mit dem Objekt verbunden bleiben. Sobald der Anwender eine Aktion mit dem Objekt durchführt, muß die Markierung aufgehoben werden. Der Anwender kann dann weitere Aktionen durchführen, die das Objekt betreffen. Deaktivieren Sie die Auswahlmarkierung auch dann, wenn der Anwender ein anderes Objekt auswählt.

Schränken Sie die Bewegungsfreiheit des Mauszeigers nicht ein. Der Mauszeiger sollte von einem Fenster zum anderen und von einer Anwendung zu anderen bewegt werden können. Bewegen Sie den Mauszeiger nicht unabhängig von der Maus, dadurch würde die physikalische Verbindung zwischen Mauszeiger und Maus unterbrochen.

Verwenden Sie den Auswahlcursor der Tastatur. Dieser Cursor markiert den Punkt, an dem Eingaben über die Tastatur erfolgen werden. So stellt das blinkende

Caret den Auswahlcursor in Textverarbeitungs-Anwendungen dar. Dieser Cursor existiert unabhängig vom Mauszeiger und ist nur über Mausklicks mit ihm verbunden.

Markieren Sie den Fokus. Der *Fokus* ist die Stelle am Bildschirm, die Eingaben über die Tastatur oder die Maus aufnimmt. Wenn der Anwender zum Beispiel ein aufspringendes Menü auswählt, besitzt dieses den Fokus. Wenn der Anwender Text optisch hervorhebt, hat dieser Text den Fokus. Markieren Sie immer das den Fokus besitzende Objekt: Ausgewählte Menüpunkte werden normalerweise invers dargestellt und angewählte Steuerelemente in einem Dialogfenster mit einer gepunktete Linie umgeben. Wenn der Anwender die Maus zieht, um eine Gruppe von Einträgen auszuwählen, wird der gesamte Bereich häufig mit einer gepunkteten Linie umgeben, die »Marque« genannt wird.

Entwerfen Sie eine Tastaturschnittstelle. Stellen Sie mit den Pfeiltasten, den Tab-Tasten und den Tastatur-Acceleratoren Alternativen zur Maus zur Verfügung. Das ist wichtig für Anwender, die keine Maus haben oder sie nicht benützen wollen.

Halten Sie sich an die Konventionen für das Klicken mit der Maus. Windows verwendet bestimmte Konventionen für das Klicken mit der Maus, die jedoch nicht zwingend sind. So bewirkt das Drücken der linken Maustaste die Auswahl und Manipulation von Objekten, das Drücken der rechten Taste führt andere Aufgaben aus, wie zum Beispiel Ändern des Modus oder Anzeigen einer Pop-up-Liste mit Optionen.

Richten Sie sich für das Betätigen der linken Maustaste nach folgenden Regeln:

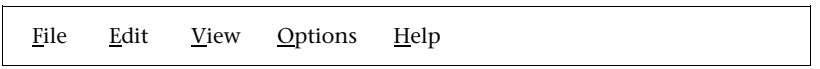
- Einfacher Klick wählt ein Objekt aus.
- Doppelter Klick wählt ein Objekt aus und aktiviert eine damit verbundene, voreingestellte oder implizierte Aktion.
- Klicken und Ziehen verändern die Größe und Position des Objekts oder wählen Text aus und markieren ihn.
- Ctrl-Taste und einfacher Klick fügen das ausgewählte Objekt zu den vorher ausgewählten Objekten hinzu.
- Shift-Taste und einfacher Klick erweitern den Bereich vorher ausgewählter Objekte zwischen dem als Anker fungierenden ersten und dem angeklickten Objekt (vgl. dazu beispielsweise den Datei-Manager).

Menüs

Windows unterstützt und erzwingt einen festumrissenen Standard für das Erscheinungsbild von Pop-up-Menüs. Trotzdem können Sie über die ausgewählten Stile und die Optionen, die darin angeboten werden sollen, relativ frei entscheiden. So sollten Sie zum Beispiel eindeutige Abkürzungen für die Tasten bzw. Tastenkombinationen vergeben, mit denen der Anwender auf Einträge in der obersten Menüzeile und in Pop-up-Menüs zugreifen kann. Einen Menüeintrag, der nach seiner Auswahl ein Dialogfenster einblendet, sollten Sie mit nachfolgenden ... markieren. Hinterlegen Sie Menüeinträge, die nicht aktiviert werden können, grau.

Die meisten Anwendungen – gleichgültig für welche Zwecke sie entworfen sind – führen immer bestimmte Operationen zur Manipulation von Dateien, zum Editieren und zum Zugriff auf ein Hilfesystem aus. Aus diesem Grund fordert CUA besondere Formate für Datei-, Edit- und Hilfemenüs. Außerdem gibt es Richtlinien für Anzeige- und Optionenmenüs.

Gemäß CUA sind die Optionen des Hauptmenüs in der folgenden Reihenfolge aufgelistet, wobei das Hilfemenü immer an letzter Stelle (das heißt, ganz rechts) steht, unabhängig davon, wieviele andere Menüauswahlpunkte angeboten werden.



Um Hilfedateien zu produzieren, die das Windows-Hilfesystem lesen kann, benötigen Sie den Help-Compiler von Microsoft.

Dialogfenster

Den vielen Möglichkeiten zum Entwurf von Dialogfenstern stehen nur wenige Richtlinien gegenüber. Sie sollten sich für eine Art entscheiden und diese dann »durchhalten«. Es gibt zwei Typen von Dialogfenstern: modale und nicht-modale. Am häufigsten werden modale Dialoge verwendet, die die Operation des Programms sozusagen einfrieren. Nicht-modale Dialoge, die die reguläre Fortsetzung des Programms erlauben, sollten Sie nur für ganz spezielle Zwecke und parallel laufende Aktivitäten einsetzen, wie zum Beispiel dann, wenn ein Dokument auf Rechtschreibfehler überprüft wird.

Für Öffnen- und Sichern-Als-Dialoge fordert CUA eine besondere Art von Dialogfenster. Das in ObjectWindows vordefinierte Datei-Dialogfenster entspricht diesen Forderungen.

Anmerkungen zu Entwürfen

Wählen Sie eine Größe für das Hauptfenster. Das Hauptfenster sollte gerade so groß sein, daß die notwendigen Informationen darin Platz haben und daß ein ausreichend großer Client-Bereich angelegt werden kann.

Seien Sie sorgfältig bei der Veränderung von Fenstergrößen. Es gibt zwei Möglichkeiten für das Neuzeichnen von Fenstern, deren Größe verändert wurde.

Sie können die Fensterinhalte in den ursprünglichen Proportionen anzeigen, wobei die Bereiche, die nicht mehr in das Fenster passen, abgeschnitten werden. Dieses Vorgehen eignet sich für sogenannte »what-you-see-is-what-you-get«-Anwendungen (WYSIWYG).

Alternativ dazu kann der Inhalt aber erneut formatiert werden, so daß er genau in das Fenster paßt. Diese Möglichkeit ist für Spiele und Utilities angebracht.

Verwenden Sie Farben sparsam. Farben sollten eine bestehende Kommunikationsverbindung mit dem Anwender unterstreichen. Die Farbgebung sollte konsistent sein und der Anwender muß die Möglichkeit zur Änderung der einzelnen Farben haben.

Sichere Programme

Die Fehlerbehandlung in Programmen mit interaktiver Anwenderschnittstelle ist weitaus komplizierter als in Kommandozeilen-Anwendungen. In einer nicht-interaktiven Anwendung gilt es als ganz normal (und wird sogar erwartet), daß Fehler zur Anzeige einer Meldung und zur Beendigung des Programms führen. Bei interaktiven Anwendungen muß das Programm aber die Fehler wieder auffinden und in einem für den Anwender akzeptablen Zustand bleiben. Das Programm darf unter keinen Umständen die Daten zerstören, die der Anwender gerade bearbeitet oder – ohne Rücksicht auf die Fehlerart – beendet werden. Ein Programm, das diese Kriterien erfüllt, kann als »sicher« gelten und so bezeichnet werden.

ObjectWindows erleichtert das Schreiben von sicheren Programmen. Es unterstützt einen Programmierstil, der das Entdecken und Auffinden von Fehlern leichter macht, vor allen den schwer in den Griff zu bekommenden »Out of memory«-Fehler.

Das Konzept der atomaren Operationen macht diese Art der Fehlerbehandlung möglich.

Programmieren unter dem Motto »Alles oder Nichts«

Eine atomare Operation ist eine Operation, die sich nicht weiter in kleinere Operationseinheiten zerlegen läßt. In unserem Zusammenhang heißt das: Eine atomare Operation schlägt entweder vollkommen fehl oder ist hundertprozentig erfolgreich. Atomare Operationen sind vor allem bei der Zuweisung von Speicherplatz sehr hilfreich.

Programme reservieren – normalerweise – Speicherplatz in kleinen »Portionen«. Wenn Sie zum Beispiel ein Dialogobjekt konstruieren, wird für das Dialogfenster und für alle anderen zum Dialogfenster gehörigen Objekte (wie untergeordnete Steuerelemente) Speicherplatz reserviert. Jede dieser Reservierungen könnte potentiell fehlschlagen. Deshalb muß jede potentielle Fehlersituation daraufhin überprüft werden, ob die nächste Reservierung durchgeführt werden kann oder ob die Weiterarbeit gestoppt werden muß. Wenn eine Reservierung nicht erfolgreich war, sollte bereits belegter Speicherplatz wieder freigegeben werden.

Es wäre ideal, wenn Sie alle Reservierungen durchführen und dann überprüfen würden, ob eine davon mißlungen ist. Führen Sie den Sicherheitsbereich ein.

Der Sicherheitsbereich

ObjectWindows legt einen bestimmten Speicherplatzumfang (8 KByte) an, der *Sicherheitsbereich* genannt wird. Wenn das Reservieren von Speicherplatz im Heap den Sicherheitsbereich berührt, liefert die Systemfunktion *LowMemory* den Wert *True* zurück. Das bedeutet, daß weitere Reservierungen nicht sicher sind und unter Umständen fehlschlagen können.

Damit der Sicherheitsbereich seine Aufgabe erfüllen kann, muß er mindestens so groß sein, wie die größte atomare Zuweisung. Mit anderen Worten: Der Bereich muß groß genug sein, daß alle Reservierungen zwischen den Überprüfungen mit *LowMemory* erfolgreich sind; dafür reichen normalerweise 8 KByte aus.

Wenn Ihre Anwendung zum ersten Mal aktiv wird und das Haupt- und die untergeordneten Fenster erstellt, überprüft sie auch gleich, ob genügend Speicherplatz zur Verfügung steht. Wenn das Programm jedoch nach der Anzeige des Hauptfensters Fenster oder Dialoge als dynamische Objekte erstellt, müssen Sie explizit den Mechanismus zur Speicherplatz-Überprüfung aufrufen. Dies ist erforderlich, wenn Dialoge oder Pop-up-Fenster als Antwort auf eine Menüauswahl erzeugt werden. MDI-Anwendungen überprüfen auch bei der Erstellung von untergeordneten Fenstern, ob genügend Speicherplatz zur Verfügung steht.

Wenn bei der Erstellung von Fenstern oder nicht-modalen Dialogen auf ausreichenden Speicherplatz überprüft werden soll, rufen Sie anstelle der *Create*-Methode *MakeWindow* auf, eine *TApplication*-Methode, die von Ihrer Anwendung geerbt wurde.

Die folgende Methode erstellt ein Fenster, ohne zu überprüfen, ob genügend Speicherplatz vorhanden ist:

```
procedure TMywindow.Help(var Msg: TMessage);
var
  HelpWnd: PHelpWindow;
begin
  HelpWnd := New(PHelpWindow, Init(@Self, 'Help System'));
  HelpWnd^.Create;
end;
```

Die folgende Methode überprüft bei der Fenstererstellung, ob genügend Speicherplatz zur Verfügung steht:

```
procedure TMyWindow.Help(var Msg: TMessage);
var
  HelpWnd: PHelpWindow;
begin
  HelpWnd := Application^.MakeWindow(New(PHelpWindow,
    Init(@Self, 'Help System')));
  if HelpWnd <> nil then  { das Fenster wurde erfolgreich erstellt }
end;
```

Verwenden Sie zur Bezeichnung des Anwendungsobjekts die globale Variable *Application*, die für jede ObjectWindows-Anwendung definiert ist. *MakeWindow* liefert entweder einen Zeiger auf das Fensterobjekt oder nil zurück. Letzteres ist der Fall, wenn die Erstellung nicht erfolgreich ausgeführt werden konnte.

Wenn Sie bei der Ausführung von modalen Dialogen überprüfen wollen, ob genügend Speicherplatz vorhanden ist, rufen Sie anstelle der Dialogmethode *Execute* die zum Anwendungstyp gehörige Methode *ExecDialog* auf.

```
procedure TTestWindow.RunDialog(var Msg: TMessage);
var
  TestDlg: PTestDialog;
  RetValue: Integer;
begin
  TestDlg := New(PTestDialog, Init(@Self, 'DIAL1'));
  RetValue := Application^.ExecDialog(TestDlg);
  if RetValue = id_Cancel then
    { Dialog wurde nicht erstellt oder vom Benutzer abgebrochen }
if RetValue = id_Cancel then  { Dialog wurde vom Benutzer abgebro-
chen }
if RetValue < 0 then         { Dialog konnte nicht erstellt werden }

end;
```

Bei erfolgreicher Durchführung liefert *ExecDialog* den Rückgabewert des Dialogs, normalerweise eine 0 oder einen positiven Integerwert. Wenn die Ausführung des Dialogs nicht erfolgreich war, liefert *ExecDialog* eine (der wenigen) Fehlerkonstanten in Form eines negativen Integerwerts. Dies ist wichtig zu wissen, weil eine fehlerhafte Dialogausführung automatisch dazu führt, daß das Dialogobjekt freigegeben wird. Überprüfen Sie vor der Freigabe des Dialogobjekts, ob der Rückgabewert negativ war.

MakeWindow und *ExecDialog* benachrichtigen das Anwendungsobjekt, wenn zu wenig Speicherplatz vorhanden ist. Dazu wird die *Error*-Methode des Anwendungsobjekts mit der Konstante *em_OutOfMemory* als Argument aufgerufen. Die *Error*-Methode, die von *TApplication* abgeleitet ist, zeigt ein Meldungsfenster an, das den Anwender auf die Fehlersituation aufmerksam macht.

Weitere Fehler bei der Erstellung von Fenstern

Neben nicht ausreichendem Speicherplatz gibt es weitere Bedingungen, die die Erstellung und korrekte Anzeige eines Fensters oder eines Dialogs verhindern. Microsoft Windows läßt die Erstellung eines Schnittstellenelements nicht zu, wenn nicht ausreichend Speicherplatz zur Verfügung steht oder wenn falsche Werte an eine Windows-Funktion übergeben wurden. Die Anwendung muß über diese Fehler (zum Beispiel von zu wenig Speicherplatz) in Kenntnis gesetzt werden.

Die ObjectWindows-Methoden, die Schnittstellenobjekte konstruieren und erstellen, melden die oben erwähnten Fehler. *TWindowObject* definiert das Integerfeld *Status*, dessen Wert 0 ist, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers enthält das Feld einen der in Tabelle 19.1 aufgeführten Werte.

Wenn Sie überprüfen wollen, ob die Fenster-Konstruktoren oder -Erstellungsmethoden Fehler enthalten, besetzen Sie *Status* mit einer der oben genannten

oder einer eigenen Konstante. Wenn ein Fenster zum Beispiel eine Datei öffnen muß, dabei aber nicht erfolgreich ist, kann es in Abhängigkeit von einer Konstante wie *em_FileOpenErorr* eine Fehlermeldung ausgeben.

Wenn *TWindow.Create* oder *TDialog.Execute* – aufgerufen von *MakeWindow* oder *ExecDialog* – Werte zurückliefern, die nicht 0 sind, meldet die aufrufende Methode den Fehler an die *Error*-Methode des Fenstertyps. Diese ruft per Voreinstellung die *Error*-Methode des Anwendungstyps auf. Sie können die *Error*-Methode des Fenster- oder Anwendungstyps aber überschreiben, wenn Sie jeden Fehler gesondert behandeln möchten:

```
constructor EditorWindow.Init(AParent: PWindowsObject; ATitle:
  PChar; AFileName: PChar):
begin
  TWindow.Init(AParent, ATitle);
  if (Status >= 0) and not OpenTheFile(AFileName) then
    Status := em_FileOpenError;
end;:
```

Tabelle 19.1 Werte für Fehler bei der Fenstererstellung

Fehler	Bedeutung
e*m_OutOfMemory*	Die Speicherreservierung hat den Sicherheitsbereich berührt.
em_InvalidWindow	Das Fensterelement konnte nicht erstellt werden.
em_InvalidClient	Ein MDI-Fensterobjekt konnte sein Client-Fenster nicht erstellen.
em_InvalidChild	Ein oder mehrere untergeordnete Fenster des Objekts haben einen Fehler erzeugt.
em_InvalidMainWindow	Das Hauptfenster konnte nicht angelegt werden.

Großverbraucher

Sie sollten auch überprüfen, ob genügend Speicherplatz für die Großverbraucher zur Verfügung steht. Großverbraucher sind Fensterobjekte, die Speicherplatz reservieren, dessen Umfang größer ist als der Sicherheitsbereich. Dies ist zum Beispiel bei Objekten der Fall, die den gesamten Inhalt einer Datei in den Speicher laden. Großverbraucher sollten selbst die Speicherplatzüberprüfung durchführen und nicht warten, bis sie die Konstruktion beendet haben und *MakeWindow* dann die Überprüfung für sie vorninmt.

Wenn ein Großverbraucher keinen Speicherplatz mehr zur Verfügung hat, während er sich gerade selbst konstruiert, sollte er das *Status*-Flag mit einem Fehlerwert besetzen und nicht versuchen, weiteren Speicherplatz zu reservieren. Das *Status*-Flag wird dann in der *MakeWindow*-Methode überprüft, der Fehler gemeldet und das Fensterobjekt freigegeben.

20. Kollektionen

Pascal-Programmierer verbringen traditionellerweise sehr viel Zeit mit der Erstellung von Code, der Datenstrukturen manipuliert und verwaltet, wie zum Beispiel verkettete Listen und dynamisch in der Größe veränderbare Arrays. Das bedeutet, daß derselbe Code für ähnliche Datenstrukturen immer und immer wieder neu geschrieben und getestet werden muß.

So leistungsstark das herkömmliche Pascal auch ist, stellt es Ihnen nur vordefinierte Record- und Array-Typen zur Verfügung. Um komplexere Strukturen müssen Sie sich selbst kümmern.

Wenn Sie zum Beispiel Daten in einem Array speichern wollen, müssen Sie Code schreiben, der das Array erstellt, Daten in das Array importiert, Daten für die Bearbeitung aus dem Array extrahiert und eventuell Daten mit Ein-/Ausgabe-Schnittstellen austauscht. Benötigt das Programm später einen neuen Array-Elementtyp, müssen Sie wieder von vorne beginnen.

Wäre es da nicht eine große Hilfe, wenn ein Array-Typ mit Code versehen wäre, der die normalerweise mit einem Array durchzuführenden Operationen verwaltet? Was würden Sie zu einem Array-Typ sagen, der außerdem ohne Beeinträchtigung des Originalcodes erweitert werden kann?

Genau das ist das Ziel des ObjectWindows-Typs *TCollection*. Dieser Typ ist ein Objekt, das eine Kollektion von Zeigern speichert und Methoden für ihre Bearbeitung zur Verfügung stellt.

Kollektion-objekte

Eine Kollektion ist ein Objekt und verfügt deshalb auch über Methoden. Dazu kommen zwei weitere Eigenschaften, die bei einfachen Pascal-Arrays fehlen – eine Kollektion kann dynamisch vergrößert werden und sie ist polymorph.

Kollektionen werden dynamisch vergrößert

Die Größe eines Pascal-Standard-Arrays ist zur Compilierzeit fest. Das ist auch gut so, wenn Sie genau wissen, wie groß das Array zu jeder Zeit sein muß. Die

zur Laufzeit benötigte Größe ist aber nicht immer vorhersehbar. Wenn die Größe eines Arrays geändert werden soll, muß der Code umgeschrieben und erneut compiliert werden.

Bei einer Kollektion dagegen legen Sie einen Anfangswert fest, der zur Laufzeit dynamisch vergrößert werden kann, um ihn an die enthaltenen Daten anzugleichen. Dies erhöht die Flexibilität der compilierten Anwendung. Denken Sie jedoch daran, daß Kollektionen nicht verkleinert werden können. Sie müssen deshalb darauf achten, daß sie nicht zu umfangreich werden.

Kollektionen sind polymorph

Die Tatsache, daß jedes Element in einem Array von demselben Typ sein muß, ist eine weitere Einschränkung für Ihre Anwendung. Der Typ muß feststehen, wenn der Code compiliert wird.

Kollektionen umgehen diese Einschränkung, indem Sie typenlose Zeiger verwenden. Aber nicht genug damit. Eine Kollektion kann sogar aus Objekten (und Nicht-Objekten) von unterschiedlichem Typ und Größe bestehen, denn eine Kollektion muß nichts über die Objekte wissen, mit denen sie umgeht. Sie nimmt Objekte nur auf und gibt sie auf Anforderung wieder zurück.

Typenprüfung und Kollektionen

Eine Kollektion umgeht die strenge Typenprüfung von Pascal. Das bedeutet, daß Sie alles in eine Kollektion einbringen können. Wenn Sie wieder etwas entnehmen wollen, kann der Compiler nicht überprüfen, was Sie zu entnehmen beabsichtigen. Sie können ein *PHirtenhund* einbringen und dieses als *PSchaf* entnehmen – die Kollektion hat keinerlei Möglichkeit, Sie darüber zu informieren.

Als Turbo-Pascal-Programmierer wird Sie diese Umgehung der Typenprüfung sicher beunruhigen. Die strenge Pascal-Typenprüfung bewahrt Sie davor, nach solch hinterhältigen Fehlern suchen zu müssen. Darum sollten Sie auch vorsichtig sein: Unterschätzen Sie nicht die Schwierigkeiten bei der Suche nach Fehlern, die durch Vermischung von Typen verursacht werden, diese hat bisher der Compiler für Sie gefunden. Wenn das Programm hängt oder abstürzt, sollten Sie immer die Typen überprüfen, die in Kollektionen eingebracht und von dort wieder entnommen werden.

Kollektionen mit Nicht-Objekten

Sie können in Kollektionen auch Elemente einbringen, die keine Objekte sind. Dabei müssen Sie aber ebenfalls sehr umsichtig zu Werke gehen. Kollektionen erwarten als Eingabe immer typenlose Zeiger auf Irgendetwas. Aber einige Methoden von *TCollection* arbeiten nur mit einer Kollektion von Instanzen, die von *TObject* abgeleitet sind. Dazu gehören die Stream-Zugriffsmethoden *PutItem* und *GetItem* ebenso wie die Standardprozedur *FreeItem*.

So können Sie zum Beispiel ein PChar in einer Kollektion speichern. Wenn Sie aber versuchen, diese Kollektion an einen Stream zu schicken, werden die Ergebnisse nicht befriedigend sein – außer Sie überschreiben die Standardmethoden *GetItem* und *PutItem*. Ähnlich ist es, wenn Sie versuchen, die Kollektion freizugeben. Dabei wird versucht, mit *FreeItem* jeden Eintrag freizugeben. Wenn Sie Einträge, die nicht von *TObject* abgeleitet sind, in eine Kollektion einbringen wollen, müssen Sie die Bedeutung von »item« in *GetItem*, *PutItem* und *FreeItem* neu definieren. Diese Neudefinition führt z.B. *TStrCollection* durch.

Wenn Sie also Kollektionen mit der zu Gebote stehenden Vorsicht einsetzen, werden Sie sehen, daß Kollektionen (und die von ihnen abgeleiteten Kollektionen) schnelle, flexible und der Situation genau entsprechende Datenstrukturen sind.

Erstellen einer Kollektion

Das Erstellen einer Kollektion ist genauso so einfach wie das Definieren des Datentyps, der in die Kollektion eingebracht werden soll. Nehmen wir an, Sie sind ein Unternehmensberater und Sie wollen Rechnungsnummer, Name und Telefonnummer Ihrer Kunden speichern und wieder abrufen. Zu diesem Zweck definieren Sie zuerst das Kunden-Objekt, das in der Kollektion gespeichert werden soll.

```
type
  PClient = ^TClient;
  TClient = object(TOBject)
    Account, Name, Phone: PChar;
    constructor Init(NewAccount, NewName, NewPhone: PChar);
    destructor Done; virtual;
    procedure Print; virtual;
end;
```

Hinweis: Vergessen Sie nicht, für jeden neuen Objekttyp einen Zeiger zu definieren.

Als nächstes implementieren Sie *Init* und *Done*, um Speicher für die Kundendaten zu reservieren und wieder freizugeben, und eine *Print*-Methode, um die Daten in tabellarischer Form auszugeben. Beachten Sie, daß die Objektfelder vom Typ PChar sind, so daß nur für den tatsächlich verwendeten String Speicherplatz reserviert wird. Die Funktionen *StrNew* und *StrDispose* verwalten die dynamischen Strings auf die effizienteste Art und Weise.

```
constructor TClient.Init(NewAccount, NewName, NewPhone: PChar);
begin
  Account := StrNew(NewAccount);
  Name := StrNew(NewName);
  Phone := StrNew(NewPhone);
end;
```

```
destructor TClient.Done;
begin
  StrDispose(Account);
  StrDispose(Name);
  StrDispose(Phone);
end;

procedure TClient.Print;
begin
  Writeln(' ',
    Account, '':10 - StrLen(Account),
    Name, '':20 - StrLen(Name),
    Phone, '':16 - StrLen(Phone)),
end;
```

TClient.Done wird für jeden Kunden aufgerufen, wenn Sie die gesamte Kollektion freigeben. Nun instantiieren Sie eine Kollektion, die Ihre Kunden speichert und fügen den Kunden-Record darin ein. Der Hauptrumpf des Programms namens COLLECT1.PAS sieht folgendermaßen aus:

```
var
  ClientList: PCollection;

begin
  ClientList := New(PCollection, Init(10, 5));
  with ClientList^ do
  begin
    Insert(New(PClient, Init('91-100', 'Anders, Smitty',
      '(406) 111-2222')));
    Insert(New(PClient, Init('90-167', 'Smith, Zelda',
      '(800) 555-1212')));
    Insert(New(PClient, Init('90-177', 'Smitty, John',
      '(406) 987-4321')));
    Insert(New(PClient, Init('90-160', 'Johnson, Agatha',
      '(302) 139-8913')));
  end;

  PrintAll(ClientList);
  SearchPhone(ClientList, '(406)');
  Dispose(ClientList, Done);
end.
```

Die Prozeduren *SearchPhone* und *PrintAll* werden weiter hinten in diesem Kapitel erläutert.

Beachten Sie, wie einfach der Aufbau der Kollektion ist. Die erste Anweisung reserviert für ein neues *TCollection* mit dem Namen *ClientList* und einem Anfangswert von 10 Kunden Speicherplatz. Werden mehr als 10 Kunden in *ClientList* eingefügt, vergrößert sich die Kollektion um jeweils 5 Kunden. Die nächsten beiden Anweisungen erzeugen ein neues Kundenobjekt und fügen es in die Kollektion ein. Der Aufruf von *Dispose* am Ende gibt die gesamte Kollektion frei.

Sie müssen der Kollektion nirgends mitteilen, *welche* Art von Daten sie aufnimmt – jeder Eintrag ist ein Zeiger.

Iterator-methoden

Einfügen und Löschen von Einträgen sind nicht die einzigen Operationen, die allen Kollektionen von Haus aus gemeinsam sind. Sie werden beispielsweise häufig for-Schleifen schreiben, um alle Objekte in der Kollektion umzuordnen, weil Sie die Daten anzeigen oder Berechnungen mit ihnen ausführen wollen. Oder Sie wollen den ersten bzw. letzten Eintrag finden, der bestimmten Kriterien entspricht. Für diese Zwecke verfügen die Kollektionen über drei *Iterator*-Methoden, *ForEach*, *FirstThat* und *LastThat*. Jede dieser Methoden hat als einzigen Parameter einen Zeiger auf eine Prozedur oder Funktion.

Der ForEach-Iterator

ForEach hat einen Zeiger auf eine Prozedur als Parameter. Die Prozedur wiederum hat einen Parameter, der ein Zeiger auf einen in der Kollektion gespeicherten Eintrag ist. *ForEach* ruft diese Prozedur für jeden Eintrag auf und zwar in der Reihenfolge, in der die Einträge in der Kollektion angeordnet sind. Die Prozedur *PrintAll* in *Collect1* ist ein Beispiel für die Verwendung eines *ForEach*-Iterators.

```
procedure PrintAll(C: PCollection);

  procedure CallPrint(P: PClient); far;
  begin
    P^.Print;                          { Ruft die Print-Methode auf }
  end;

begin   { PrintAll }
  Writeln;
  Writeln;
  Writeln('Client list');
  C^.ForEach(@CallPrint);              { Druckt jeden Kunden }
end;
```

Für jeden Eintrag in der Kollektion, der als Parameter an *PrintAll* übergeben wird, wird die lokale Prozedur *CallPrint* aufgerufen, die die Kundenobjekt-Daten in formatierten Spalten ausgibt. Sie müssen darauf achten, welche Art von Prozedur Sie mit den Iteratoren aufrufen. Damit eine Routine von einem Iterator aufgerufen werden darf, muß sie folgende Bedingungen erfüllen (vgl. *CallPrint*):

- Die Routine muß eine Prozedur sein. Es darf sich nicht um eine Funktion oder eine Objektmethode handeln. Die Prozedur kann dann aber (wie in unserem Beispiel) eine Methode *aufrufen*.

- Es muß sich um eine lokale Routine handeln, die in die aufrufende Routine eingebettet ist.
- Die Routine muß mit der Direktive far oder mit der Compiler-Direktive *{$F+}* als Prozedur vom Typ far deklariert sein.
- Die Routine hat als einzigen Parameter einen Zeiger auf einen Eintrag in der Kollektion.

Die Iteratoren FirstThat und LastThat

Sie können aber nicht nur auf jedes Element in der Kollektion eine Prozedur anwenden, sondern Sie haben auch die Möglichkeit, ein bestimmtes Element in der Kollektion zu suchen, das den angegebenen Kriterien entspricht. Diese Aufgabe übernehmen die Iteratoren *FirstThat* und *LastThat*. Wie die Namen bereits andeuten, durchsuchen sie in jeweils entgegengesetzter Richtung eine Kollektion, bis sie einen Eintrag finden, der die Kriterien der Booleschen Funktion erfüllt, die als Argument übergeben wurde.

FirstThat und *LastThat* liefern einen Zeiger auf den ersten bzw. letzten Eintrag, der den Suchkriterien entspricht. Im folgenden Beispiel sehen Sie eine Prozedur, die mit der Methode *FirstThat* den ersten Kunden mit einer bestimmten Telefonnummer findet:

```
procedure SearchPhone(C: PCollection; PhoneToFind: PChar);
  function PhoneMatch(Client: PClient): Boolean; far;
  begin
    PhoneMatch := StrPos(Client^.Phone, PhoneToFind) <> nil;
  end;

  var
    FoundClient: PClient;

  begin                                          { SearchPhone }
    Writeln;
    FoundClient := C^.FirstThat(@PhoneMatch);
    if FoundClient = nil then
      Writeln('No client met the search requirement')
    else
    begin
      Writeln('Found client:');
      FoundClient^.Print;
  end;
end;
```

Beachten Sie, daß *PhoneMatch* lokal zu *SearchPhone* ist und das far-Aufrufmodell verwendet. In diesem Fall handelt es sich um eine Funktion, die nur dann *True* zurückliefert, wenn die Telefonnummer des Kunden und das Suchmuster übereinstimmen. Wenn kein Eintrag in der Kollektion den Suchkriterien entspricht, wird von *FirstThat* ein nil-Zeiger zurückgeliefert.

Vergessen Sie nicht: *ForEach* ruft eine anwenderdefinierte Prozedur auf, während *FirstThat* und *LastThat* eine anwenderdefinierte Boolesche Funktion aufrufen. In beiden Fällen wird der anwenderdefinierten Prozedur bzw. Funktion ein Zeiger auf ein Objekt in der Kollektion übergeben.

Sortierte Kollektionen

Es gibt Fälle, in denen Ihre Daten in einer bestimmten Reihenfolge vorliegen sollten. ObjectWindows stellt den speziellen Kollektionstyp *TSortedCollection* zur Verfügung, der es Ihnen ermöglicht, Daten in eine gewünschte Reihenfolge zu bringen.

TSortedCollection ist von *TCollection* abgeleitet und sortiert die gegebenen Objekte. Außerdem wird überprüft, ob es in der Kollektion nach Hinzufügen eines neuen Elements zu doppelten Schlüsseln kommt.

Das Boolesche Feld *Duplicates* legt fest, ob doppelte Schlüssel zugelassen sind. Ist *Duplicates* auf *False* gesetzt (was per Voreinstellung der Fall ist), ersetzt ein neu hinzugefügtes Element ein bereits in der Kollektion existierendes Element mit demselben Schlüssel. Ist *Duplicates* auf *True* gesetzt, wird das neue Element einfach in die Kollektion eingefügt.

TSortedCollection ist ein abstrakter Typ. Wenn Sie ihn verwenden wollen, müssen Sie zuerst den Typ der Daten, die in die Kollektion eingebracht werden, festlegen und zwei Methoden definieren, die die gewünschten speziellen Sortieranforderungen bestimmen. Zu diesem Zweck müssen Sie einen neuen Typ von *TSortedCollection* ableiten. In unserem Beispiel leiten wir den Typ *TClientCollection* ab.

TClientCollection weiß bereits, wie die Arbeit mit einer Kollektion aussieht. *TClientCollection* kann neue Kunden-Records einfügen und bestehende löschen – dieses Basisverhalten ist das Erbe von *TCollection*. Sie müssen *TClientCollection* lediglich mitteilen, welches Feld als Sortierschlüssel verwendet werden soll, wie zwei Kunden verglichen werden und welcher vor dem anderen in der Kollektion stehen soll.

Dazu überschreiben Sie die Methoden *KeyOf* und *Compare* und implementieren sie folgendermaßen:

```
PClientCollection = ^TClientCollection;
TClientCollection = object(TSortedCollection)
  function KeyOf(Item: Pointer): Pointer; virtual;
  function Compare(Key1, Key2: Pointer): Integer; virtual;
end;

function TClientCollection.KeyOf(Item: Pointer): Pointer;
begin
  KeyOf := PClient(Item)^.Account;
end;

function TClientCollection.Compare(Key1, Key2: Pointer):
   Integer;
```

```
begin
  Compare := StrIComp(PChar(Key1), PChar(Key2));
end;
```

Schlüssel müssen mit einem Typ versehen werden (typecast), weil sie typenlose Zeiger sind.

KeyOf definiert, welches Feld oder welche Felder als Sortierschlüssel verwendet werden sollen. In unserem Fall fungiert das Feld *Account* als Schlüssel. *Compare* hat zwei Sortierschlüssel und legt fest, welcher in der Sortierreihenfolge zuerst kommt. *Compare* liefert -1, 0 oder 1, je nachdem, ob *Key1* kleiner, gleich oder größer als *Key2* ist. In unserem Beispiel wird der Schlüsselstring (*Account*) alphabetisch unter Berücksichtigung der Groß-/Kleinschreibung sortiert. Dazu wird die Funktion *StrIComp* der Unit *Strings* aufgerufen. Sie könnten die Kollektion auch nach dem Namen anstatt nach der Rechnungsnummer sortieren, indem Sie einfach *KeyOf* das Feld *Name* zurückliefern lassen.

Da die von *KeyOf* zurückgelieferten und an *Compare* übergebenen Schlüssel typenlose Zeiger sind, müssen Sie diese in unserem Beispiel in den Typ PChar wandeln, bevor sie dereferenziert oder an *StrIComp* übergeben werden.

Das war schon alles, was an Definitionen erforderlich ist. Wenn Sie nun *ClientList* neu als *PClientCollection* anstatt als *PCollection* definieren, werden Ihre Kunden in alphabetischer Reihenfolge ausgegeben.

Sehen Sie sich das folgende Codefragment an, das Sie in COLLECT1.PAS finden:

```
var
  ClientList: PClientCollection;
  ...
begin
  ClientList := New(PClientCollection, Init(10, 5));
  ...
end.
```

String-kollektionen

Viele Programme müssen sortierte Strings verwalten. Für diesen Zweck stellt ObjectWindows die spezielle Kollektion *TStrCollection* zur Verfügung. Beachten Sie, daß die Elemente in einer *TStrCollection keine* Objekte sind, sondern Zeiger auf nullterminierte Strings. Da eine Stringkollektion von *TSortedCollection* abgeleitet ist, können Stringduplikate gespeichert werden.

Hinweis: Für das Speichern von Pascal-Strings steht *TStringCollection* zur Verfügung.

Wenn Sie eine Stringkollektion verwenden wollen, müssen Sie nur eine Zeigervariable für sie deklarieren. Weisen Sie der Kollektion Speicherplatz zu, indem Sie einen Anfangswert vergeben und den Umfang festlegen, um den die Kollektion bei Hinzufügen weiterer Strings vergrößert wird. Das folgende Beispiel hat den Namen COLLECT3.PAS.

```
var
  WordList: PCollection;
  WordRead: PChar;
  ...
begin
WordList := New(PStrCollection, Init(10, 5));
  ...
```

WordList umfaßt zu Beginn zehn Strings und kann schrittweise um jeweils fünf Strings erweitert werden – Sie müssen nur Strings einfügen. In unserem Beispiel werden Wörter aus einer Textdatei gelesen und in die Kollektion eingefügt:

```
repeat
  ...
  if GetWord(WordRead, WordFile)^ <> #0 then
    WordList^.Insert(StrNew(WordRead));
   ...
  until WordRead[0] = #0;
  ...
Dispose(WordList, Done);
```

Beachten Sie, daß die Funktion *StrNew* das eingelesene Wort kopiert und die Adresse des kopierten Wortes an die Kollektion übergibt. Wenn Sie eine Kollektion verwenden, übergeben Sie ihr die Kontrolle über die betroffenen Daten. Die Kollektion gibt am Ende der Bearbeitung den für die Daten reservierten Speicherplatz durch den Aufruf von *Done* wieder frei. *Done* gibt zuerst jedes Element der Kollektion frei und danach die Kollektion *WordList* selbst.

Noch einmal: Iteratoren

Die *ForEach*-Methode übergibt der Reihe nach jeden Eintrag der Kollektion an die von Ihnen angegebene Prozedur. Für die Fortsetzung unseres Beispiels erhält die Prozedur *PrintWord* einen Zeiger, der auf den anzuzeigenden String weist. Beachten Sie, daß *PrintWord* eine lokale Prozedur ist. Sie wird von einer anderen Prozedur namens *Print* umgeben, der ein Zeiger auf *TStrCollection* übergeben wird. *Print* verwendet die *ForEach*-Methode dazu, um jeden Eintrag in der Kollektion an die *PrintWord*-Prozedur zu übergeben.

```
procedure Print(C: PCollection);

  procedure PrintWord(P: PChar); far;
  begin
  Writeln(P);                          { Zeigt den String an }
  end;

begin  { Print }
  Writeln;
```

```
  Writeln;
  C^.ForEach(@PrintWord);              { Ruft PrintWord auf }
end;
```

PrintWord ist eine Prozedur, die einen Stringzeiger als Argument hat und seinen Wert an *Writeln* übergibt. Beachten Sie die far-Direktive nach der *PrintWord*-Deklaration. *PrintWord* darf keine Methode sein – es muß sich um eine lokale Prozedur handeln. *Print* ist die Rahmenroutine, die für jeden Eintrag in der Kollektion bestimmte Aufgaben ausführen muß, zum Beispiel die Anzeige oder Änderung von Daten. Hierfür können Sie mehrere Prozeduren wie *PrintWord* anlegen, die aber alle innerhalb von *Print* liegen und vom Typ far sein müssen (d.h. die Direktive far oder *{$F+}* verwenden).

Einträge suchen

Sortierte Kollektionen (und somit auch Stringkollektionen) verfügen über eine *Search*-Methode, die die Indexposition eines Eintrags mit einem bestimmten Schlüssel zurückliefert. Wie aber läßt sich ein Eintrag in einer Kollektion finden, die nicht sortiert ist? Was ist der Fall, wenn die Suchkriterien nicht den Schlüssel selbst betreffen? Die Antwort lautet, wie Sie sich nach den bisherigen Erläuterungen sicher denken können: Verwendung von *FirstThat* oder *LastThat*. Sie definieren einfach eine Boolesche Funktion, um die gewünschten Kriterien zu testen, und rufen dann zum Beispiel *FirstThat* auf.

Polymorphe Kollektionen

Sie haben gesehen, daß Kollektionen Daten jeden Typs dynamisch speichern können und über eine große Anzahl von Methoden für einen effizienten Zugriff auf die Daten verfügen. So definiert *TCollection* selbst 23 Methoden. Sie werden vor allem von dem Zeitverhalten angenehm überrascht sein, wenn Sie in Ihren Programmen Kollektionen einsetzen. Kollektionen wurden unter diesen beiden Aspekten – Flexibilität und Schnelligkeit – entworfen.

Aber die wahre Stärke der Kollektionen liegt in der Polymorphie: Sie können in einer Kollektion nicht nur einen Objekttyp, sondern viele verschiedene Objekttypen aus Ihrer Objekthierarchie speichern!

Wenn Sie die vorausgehenden Beispiele für Kollektionen betrachten, sehen Sie, daß die Einträge in jeder Kollektion denselben Typ haben. Es gab da beispielsweise eine Liste mit Strings, in der alle Einträge Strings sind. Aber Kollektionen können *jeden* Typ aufnehmen, der von *TObject* abgeleitet ist. Sie können die Typen mischen, wobei aber die Objekte normalerweise über Gemeinsamkeiten verfügen. Üblicherweise werden die Objekte einen abstrakten Vorfahren gemeinsam haben.

Im folgenden finden Sie ein Beispiel, das drei verschiedene grafische Objekte in eine Kollektion einfügt. Dann wird mit Hilfe des *ForEach*-Iterators die Kollektion durchquert und jedes Objekt angezeigt.

Anders als die übrigen Beispiele in diesem Kapitel verwendet COLLECT4.PAS Windows-Funktionen, um in einem Fenster zu zeichnen. Stellen Sie sicher, daß

die uses-Klausel die Units *WinProcs* und *WinTypes* enthält. Im ersten Schritt wird das Vorfahrenobjekt definiert.

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    Rect: TRect;
    constructor Init(Bounds: TRect);
    procedure Draw(DC: HDC); virtual;
end;
```

Aus dieser Deklaration ersehen Sie, daß sich jedes grafische Objekt selbst initialisieren (*Init*) und auf dem Bildschirm anzeigen (*Draw*) kann. Definieren Sie jetzt eine Ellipse, ein Rechteck und ein Tortenstück, die alle von diesem gemeinsamen Vorfahren abgeleitet sind:

```
PGraphEllipse = ^TGraphEllipse;
TGraphEllipse = object(TGraphObject)
  procedure Draw(DC: HDC); virtual;
end;

PGraphRect = ^TGraphRect;
TGraphRect = object(TGraphObject)
  procedure Draw(DC: HDC); virtual;
end;

PGraphPie = ^TGraphPie;
TGraphPie = object(TGraphObject)
  ArcStart, ArcEnd: TPoint;
  constructor Init(Bounds: TRect);
  procedure Draw(DC: HDC); virtual;
end;
```

Alle drei Objekttypen erben das *Rect*-Feld von *TGraphObject*, sind aber unterschiedlich groß. *TGraphEllipse* und *TGraphRect* erfordern andere Zeichenmethoden als *TGraphPie*. Die Methoden der beiden ersteren benötigen nur Informationen über Größe und Position, während *TGraphPie* zusätzliche Felder und einen Konstruktor erforderlich macht, um sich selbst korrekt anzuzeigen. Im folgenden sehen Sie den Code, der diese unterschiedlichen Grafikformen in die Kollektion einfügt:

```
...
GraphicsList := New(PCollection, Init(10, 5)); {Kollektion erzeugen}

for I := 1 to NumToDraw do
begin
  case I mod 3 of                                {Objekt erzeugen}
    0: P := New(PGraphRect, Init(Bounds));
    1: P := New(PGraphEllipse, Init(Bounds));
    2: P := New(PGraphPie, Init(Bounds));
  end;
```

```
    GraphicsList^.Insert(P);              { ... und in die
                                            Kollektion einfügen }
  end;

  ...
```

Die for-Schleife fügt grafische Objekte in die *GraphicsList*-Kollektion ein. Alles, was Sie über die Objekte wissen, ist, daß jedes Objekt in *GraphicsList* eine Art von *TGraphObject* ist. Wenn ein Objekt eingefügt wurde, haben Sie keinerlei Information darüber, ob es sich um ein Rechteck, eine Ellipse oder ein Tortenstück handelt.

Dank der Polymorphie müssen Sie darüber auch nicht auf dem laufenden sein, weil jedes Objekt die Daten und den Code enthält (*Draw),* die es braucht. Durchqueren Sie die Kollektion mit einer Iterator- Methode und lassen Sie jedes Objekt sich selbst anzeigen:

```
procedure DrawAll(C: PCollection);

  procedure CallDraw(P: PGraphObject); far;
  begin
    P^.Draw(PaintDC);                 { Draw-Methode aufrufen }
  end;

begin    { DrawAll }
  C^.ForEach(@CallDraw);              { Alle Objekte zeichnen }
end;

var
  GraphicsList: PCollection;
begin
  ...
  if GraphicsList <> nil then DrawAll(GraphicsList);
  ...
end.
```

Die Fähigkeit einer Kollektion, unterschiedliche, aber miteinander verwandte Objekte zu speichern, gehört zu den leistungsstarken Eckpfeilern objektorienterter Porgrammierung. Im nächsten Kapitel stellen wir Ihnen dieselben Polymorphie-Prinzipien im Zusammenhang mit Streams vor.

Kollektionen und Speicherverwaltung

Ein *TCollection*-Kollektion kann dynamisch von ihrer Anfangsgröße, die mit *Init* angelegt wurde, bis zu einem maximalen Umfang von 16380 Elementen anwachsen. Die maximale Kollektionsgröße wird von ObjectWindows in der Variablen *MaxCollectionSize* gespeichert. Jedes Element, das Sie in die Kollektion einfügen, belegt nur vier Bytes, da es als Zeiger gespeichert wird.

Keine Bibliothek mit dynamischen Datenstrukturen wäre ohne Vorsorge für

Fehlersituationen vollständig. Steht nicht genügend Speicher für die Initialisierung einer Kollektion zur Verfügung, wird ein nil-Zeiger zurückgeliefert.

Wenn zur Laufzeit beim Hinzufügen eines Elements nicht mehr ausreichend Speicherplatz vorhanden ist, wird die Methode *TCollection.Error* aufgerufen und ein Heap-Zuweisungsfehler ausgelöst. Sie können *TCollection.Error* überschreiben und Ihre eigenen Fehlermeldungen und -behandlungen verwenden.

Sie müssen der Verfügbarkeit des Heap besondere Aufmerksamkeit zuwenden, da der Anwender über ein ObjectWindows-Programm mehr Kontrolle als über ein traditionelles Pascal-Programm ausübt. Wenn der Anwender das Hinzufügen von Objekten in eine Kollektion steuert (zum Beispiel durch Öffnen eines neuen Fensters), kann nicht abgeschätzt werden, ob es zu einem Heap-Fehler kommen könnte. Sie sollten den Anwender vor schweren Laufzeitfehlern schützen, indem Sie selbst den verfügbaren Speicher überprüfen und im Falle eines möglichen Überlaufs den Anwender in Kenntnis setzen.

21. Streams

ObjectWindows bietet zusammen mit objektorientierten Programmiertechniken leistungsstarke Möglichkeiten für die Kapselung von Code und Daten sowie für den Aufbau von miteinander in Beziehung stehenden Objektstrukturen. Aber wie sieht es aus, wenn Sie ganz simple Aufgaben ausführen wollen, wie zum Beispiel das Speichern von Objekten auf Platte/Diskette?

Wenn Daten einfach in einem Record stehen, ist das Schreiben auf Platte/Diskette ein klar umrissener Vorgang. Daten in einem ObjectWindows-Programm sind dagegen eng mit Objekten verknüpft. Sie können selbstverständlich die Daten von den Objekten trennen und Sie dann in eine Datei auf Platte/Diskette speichern. Die Verbindung von Daten und Objekten bietet aber so viele Vorteile, daß die Trennung einen Rückschritt bedeuten würde.

Können nun OOP und ObjectWindows selbst etwas zur Lösung dieses Problems beitragen? Sie können – und zwar mit Hilfe der Streams!

Ein Stream in ObjectWindows ist eine Kollektion von Objekten, die zu einem Ziel »unterwegs ist«: zu einer Datei, in das EMS, zu einem seriellen Anschluß oder zu einer anderen Geräteschnittstelle. Streams behandeln Ein- und Ausgaben auf der Objekt- und nicht auf der Datenebene. Wenn Sie ein ObjectWindows-Objekt erweitern, müssen Sie die Behandlung der zusätzlichen Datenfelder definieren. Die Streams übernehmen die Ausführung dieser komplexen Vorgänge für Sie.

Die Aufgabe: Objekt-Ein-/Ausgaben

Als Pascal-Programmierer wissen Sie, daß vor jeder Datei-Ein-/Ausgabe dem Compiler mitgeteilt werden muß, welche Datentypen aus einer Datei gelesen oder in eine Datei geschrieben werden sollen. Die Datei muß typisiert sein und der Typ muß zur Compilierzeit feststehen.

Turbo Pascal verfügt über Mittel zur Umgehung dieser Regel: den Zugriff auf nicht-typisierte Dateien mit *BlockWrite* und *BlockRead*. Die fehlende Typenprüfung verlangt vom Programmierer jedoch ein hohes Maß an Verantwortung, obwohl andererseits sehr schnelle binäre Ein- und Ausgaben durchgeführt werden können.

Ein zweites Problem liegt darin, daß Sie Dateien nicht direkt mit Objekten

verwenden können. Turbo Pascal erlaubt es nicht, eine typisierte Datei mit Objekten zu erstellen. Da Objekte virtuelle Methoden beinhalten können, deren Adressen zur Laufzeit bestimmt werden, ist das Speichern der VMT-Information außerhalb des Programms ebenso sinnlos, wie das Einlesen dieser Information in ein Programm.

Wie gesagt: Das Problem läßt sich umgehen. Sie können die Daten aus Ihren Objekten herauskopieren und die Information in einer beliebigen Datei speichern und dann die Objekte aus den »Rohdaten« wieder aufbauen. Dies ist sicherlich nicht die eleganteste Art und Weise, die zudem die Konstruktion von Objekten nicht gerade vereinfacht.

Die Lösung: Streams

Mit ObjectWindows lassen sich beide Probleme lösen, wobei Ihnen noch weitere Vorteile zugute kommen. Streams geben Ihnen einfache, aber elegante Mittel an die Hand, mit deren Hilfe Sie Objektdaten außerhalb des Programms speichern können.

Streams sind polymorph

Ein ObjectWindows-Stream bietet Ihnen die Vorteile von typisierten und nicht-typisierten Dateien: die Typenprüfung ist weiterhin vorhanden, aber was Sie an einen Stream senden, muß nicht zur Compilierzeit festgelegt sein. Der Grund dafür liegt darin, daß Streams wissen, daß sie es mit Objekten zu tun haben – solange das Objekt von *TObject* abgeleitet ist. Unterschiedliche ObjectWindows-Objekte können ebenso einfach in denselben Stream geschrieben werden wie ein Gruppe von identischen Objekten.

Streams bearbeiten Objekte

Sie müssen lediglich definieren, welche Objekte ein Stream bearbeiten soll. Der Stream weiß dadurch, wie er die Daten mit den VMTs zu vergleichen hat. Dann können Sie die Daten in den Stream einbringen und genauso einfach wieder daraus entnehmen.

Wie schafft es nun ein Stream, solch unterschiedliche Objekte wie ein *TCollection* und ein *TDialog* zu lesen und zu speichern und zur Compilierzeit nicht wissen zu müssen, welche Objekte bearbeitet werden? Hierin liegt der große Unterschied zur traditionellen Behandlung von Ein- und Ausgaben in Pascal. Ein Stream kann sogar neue Objekttypen bearbeiten, die noch nicht einmal erstellt waren, als der Stream compiliert wurde.

Die Antwort auf die obige Frage lautet: *Registrierung*. Jedem ObjectWindows-

Objekttyp (und allen neuen Objekttypen, die Sie aus der Hierarchie ableiten) ist eine eindeutige Registrierungsnummer zugewiesen. Diese Nummer wird vor den Objektdaten in den Stream geschrieben. Wenn die Daten dann aus dem Stream gelesen werden, erhält ObjectWindows als erstes diese Nummer. Aufgrund dieser Information weiß ObjectWindows dann, wieviele Daten gelesen werden müssen und welche VMT mit den Daten verknüpft werden muß.

Der Einsatz von Streams

Im Prinzip sind Streams mit Pascal-Dateien vergleichbar. Denn eine Pascal-Datei kann im Grunde als einfache sequentielle Ein-/Ausgabeschnittstelle betrachtet werden: In die Datei wird geschrieben und aus ihr wird wieder gelesen. Ein Stream ist dementsprechend eine *polymorphe*, sequentielle Ein-/Ausgabe-Schnittstelle. Mit anderen Worten: Ein Stream verhält sich wie eine sequentielle Datei, wobei aber verschiedene Objekttypen gespeichert und gelesen werden können.

Streams können ebenso – wie Pascal-Dateien – als Ein-/Ausgabe-Schnittstellen mit Direktzugriff angesehen werden: Sie suchen eine bestimmte Position in der Datei, lesen von dieser Position oder schreiben an diese Position, liefern die Position des Dateizeigers zurück, etc. Diese Operationen können auch mit Streams ausgeführt werden, und sind im Abschnitt »Streams mit Direktzugriff« weiter hinten in diesem Kapitel beschrieben.

Es gibt zwei Aspekte bei der Verwendung von Streams, mit denen Sie sich auseinandersetzen müssen: Aufbau eines Streams und Lesen aus und Schreiben in Streams.

Streams aufbauen

Um einen Stream verwenden zu können, müssen Sie nichts weiter tun, als ihn zu initialisieren. Die genaue Syntax des *Init*-Konstruktors hängt von dem jeweiligen Typ des verwendeten Streams ab. Wenn Sie beispielsweise einen DOS-Stream öffnen, müssen Sie den Namen der DOS-Datei und den Zugriffsmodus (Nur-Lesen, Nur-Schreiben oder Lesen/Schreiben) für die Datei angeben, die den Stream enthält.

Zur Initialisierung eines gepufferten DOS-Streams, der ein Kollektionsobjekt in ein Programm lädt, müssen Sie folgenden Code schreiben:

```
var
  SaveFile: TBufStream;
begin
  SaveFile.Init('COLLECT.DTA', stOpen, 1024);
  ...
```

Sobald der Stream initialisiert ist, können Sie ihn benutzen.

TStream ist ein abstrakter Stream-Mechanismus. Sie können deshalb keine Instanz von ihm erzeugen. Aber alle Stream-Objekte sind von diesem Typ abgeleitet. Dazu gehören *TDosStream* für Datei-Ein-/Ausgaben, *TBufStream* für gepufferte Datei-Ein-/Ausgaben (für das Lesen oder Schreiben von vielen kleinen Einheiten von oder auf Platte/Diskette) und *TEmsStream*, ein Stream, der Objekte in den EMS-Speicher schreibt.

ObjectWindows implementiert außerdem noch einen indizierten Stream mit einem Zeiger auf eine Position im Stream. Durch die Positionierung des Zeigers erhalten Sie direkten Zugriff auf den Stream.

Streams lesen und schreiben

Das Basis-Stream-Objekt *TStream* implementiert drei Methoden: *Get*, *Put* und *Error*. *Get* und *Put* entsprechen in etwa den *Read*- und *Write*-Prozeduren, die Sie bei normalen Ein-/Ausgabe-Operationen verwenden. *Error* ist eine Prozedur, die bei Auftreten von Stream-Fehlern aufgerufen wird.

Die Methode Put

Sehen wir uns zunächst die Prozedur *Put* an. Die generelle Syntax für *Put* lautet:

```
SomeStream.Put(PSomeObject);
```

SomeStream ist ein von *TStream* abgeleitetes Objekt, das initialisiert wurde. *PSomeObject* steht für einen Zeiger auf ein von *TObject* abgeleitetes Objekt, das mit dem Stream registriert wurde. Diese Angaben reichen für *Put* aus. Der Stream erfährt von der VMT des *PSomeObject*, um welchen Objekttyp es sich handelt (vorausgesetzt, der Typ wurde registriert). So weiß der Stream, welche ID-Nummer und wieviele Daten danach zu speichern sind.

Für Sie als ObjectWindows-Programmierer ist von besonderem Interesse, daß *Put* auch eventuell vorhandene untergeordnete Fenster zusammen mit einer Objektgruppe in den Stream schreibt. Sie können den vollständigen Status eines Dialogs speichern, indem Sie einfach das Dialogobjekt in den Stream schreiben. Wenn Sie Ihr Programm erneut starten und den Dialog aus dem Stream einlesen, hat er denselben Status wie zum Zeitpunkt des Speicherns.

Die Methode Get

Das Zurückholen von Objekten aus einem Stream ist kein Problem. Sie müssen nur die Stream-Funktion *Get* aufrufen:

```
PSomeObject := SomeStream.Get;
```

SomeStream ist ein initialisierter ObjectWindows-Stream und *PSomeObject* ist ein Zeiger auf einen ObjectWindows-Objekttyp. *Get* liefert einen Zeiger auf das Objekt, das aus dem Stream geholt werden soll.

Wieviele Daten aus dem Stream geholt wurden und welche VMT den Daten zugeordnet ist, wird nicht von dem *PSomeObject*-Typ, sondern von dem Objekttyp im Stream bestimmt. Wenn also das Objekt an der aktuellen Position

von *SomeStream* nicht denselben Typ hat wie *PSomeObject*, erhalten Sie nicht die erwarteten Daten.

Ebenso wie *Put* bearbeitet auch *Get* komplexe Objekte. Ist das Objekt, das Sie aus dem Stream holen, ein Fenster mit untergeordneten Fenstern, so werden diese untergeordneten Fenster ebenfalls geladen.

Die Methode Error

Die *Error*-Prozedur legt fest, was beim Auftreten eines Stream-Fehlers erfolgen soll. Per Voreinstellung belegt *TStream.Error* zwei Felder im Stream: *Status* und *ErrorInfo*. Wenn Sie einer Fehlersituation anders begegnen möchten – zum Beispiel einen Laufzeitfehler generieren oder ein Fehlermeldungsfenster einblenden – müssen Sie nur die *Error*-Methode überschreiben.

Streams entfernen

Sobald die Arbeit mit einem String beendet ist, rufen Sie dessen *Done*-Methode auf, so wie Sie bei einer Datei normalerweise *Close* aufrufen würden.

```
Dispose(SomeStream, Done);
```

Das Stream-Objekt wird freigegeben und entfernt.

Objekte für Streams

Alle ObjectWindows-Standardobjekte können mit Streams verwendet werden und alle ObjectWindows-Streams kennen diese Standardobjekte. Wenn Sie einen neuen Objekttyp von einem Standardobjekt ableiten, kann dieser auf einfache Weise für den Einsatz mit Streams vorbereitet werden. Ebenso werden die Streams von der Existenz des neuen Objekttyps in Kenntnis gesetzt.

Die Methoden Load und Store

Zwei Methoden führen das Lesen und Schreiben von Objekten aus bzw. in Streams durch: *Load* und *Store*. Jedes Objekt muß über diese beiden Methoden verfügen, um für Streams einsetzbar zu sein. Trotzdem brauchen Sie die Methoden nicht direkt aufzurufen. (Sie werden von *Get* und *Put* aufgerufen.) Sie müssen nur sicherstellen, daß Ihr Objekt weiß, wie es sich selbst an einen Stream schickt, wenn es dazu aufgefordert wird.

In OOP ist diese Aufgabe sehr einfach zu lösen, da die meisten Mechanismen vom Vorfahrenobjekt vererbt werden. Ihr Objekt muß nur die Teile von sich selbst laden und speichern, die Sie hinzugefügt haben, der Rest wird durch den Aufruf von Methoden des Vorfahrenobjekts erledigt.

Nehmen wir an, Sie leiten ein neues Fenster von *TWindow* ab und benennen es nach dem Surrealisten Magritte, der viele beeindruckende Bilder von Fenstern gemalt hat:

```
type
  TMagritte = object(TWindow)
    Surreal: Boolean;
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    ...
  end;
```

Dem Datenbereich des Fensters wurde nur ein Boolesches Feld hinzugefügt. Um das Objekt laden zu können, lesen Sie einfach ein *TWindow*-Standardobjekt und lesen dann ein dem Booleschen Feld entsprechendes zusätzliches Byte ein. Dieselben Arbeitsschritte führen Sie für das Speichern des Objekts aus: Sie schreiben ein *TWindows*-Standardobjekt mit einem weiteren Byte. Die typischen *Load*- und *Write*-Methoden für abgeleitete Objekte haben folgendes Aussehen:

```
constructor TMagritte.Load(var S: Stream);
begin
  TWindow.Load(S);                    { Vorgänge laden }
  S:Read(Surreal, SizeOf(Surreal));   { Zusätzliche Felder lesen }
end;

procedure TMagritte.Store(var S: Stream);
begin
  TWindow.Store(S);                   { Vorgänger schreiben }
  S.Write(Surreal, SizeOf(Surreal)); { Zusätzliche Felder schreiben }
end;
```

Warnung: Sie müssen gewährleisten, daß der Datenumfang, der gespeichert wird, genauso groß ist wie derjenige, der geladen wird. Außerdem müssen Sie dafür sorgen, daß die Daten in der Reihenfolge geladen werden, in der sie gespeichert sind. Der Compiler liefert keine Fehler zurück. Dadurch kann es zu mehr oder weniger schweren Fehlern kommen. Wenn Sie ein Objektfeld ändern, müssen die Methoden *Load* und *Store* aktualisiert werden.

Stream-Registrierung

Sie müssen für ein neues Objekt nicht nur die Methoden *Load* und *Store* definieren, sondern auch den neuen Objekttyp für die Verwendung mit Streams registrieren. Dazu führen Sie zwei Arbeitsschritte aus: Definieren Sie einen Stream-Registrierungs-Record und übergeben Sie ihn an die globale Prozedur *RegisterType*.

Hinweis: ObjectWindows registriert alle Standardobjekte, so daß Sie nur die neu definierten Objekte registrieren müssen.

Bei der Definition eines Registrierungs-Records müssen Sie sich an das folgende Format halten. Stream-Registrierungs-Records sind Pascal-Records vom Typ *TStreamRec*, die folgendermaßen definiert werden:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

Per Konvention erhalten alle Registrierungs-Records von ObjectWindows denselben Namen wie der entsprechende Objekttyp, wobei am Beginn des Namens ein »R« anstelle des »T« steht.

So lautet zum Beispiel der Registrierungs-Record für *TCollection RCollection* und für *TMagritte RMagritte*. Abstrakte Typen, wie *TObject* und *TWindowsObject*, haben keine Registrierungs-Records, da nie Instanzen dieser Typen in Streams gespeichert werden.

ID-Nummern von Objekten

Das Feld *ObjType* ist der einzige Teil des Records, den Sie berücksichtigen müssen, der Rest wird automatisch verwaltet. Jeder neue Typ, den Sie definieren, muß eine eigene, eindeutige Typ-Bezeichnernummer haben.

ObjectWindows reserviert für Standardobjekte 0 bis 99 als Registrierungsnummern, so daß Ihnen die Zahlen 100 bis 65535 als Nummern zur Verfügung stehen.

Wichtig: Es liegt bei Ihnen, eine Bibliothek mit ID-Nummern für alle neuen Objekte, die in einem Ein-/Ausgabe-Stream verwendet werden sollen, anzulegen und zu verwalten und die IDs den Anwendern der Units zur Verfügung zu stellen. Wie bei Menü-IDs und anwenderdefinierten Meldungen können Sie beliebige Nummern zuweisen, solange diese in dem angegebenen Bereich eindeutig sind.

Automatische Felder

Das Feld *VmtLink* stellt die Verbindung zu der virtuellen Methodentabelle (VMT) des Objekts her. Sie weisen es als Offset Ihres Objekttyps zu:

```
RSomeObject.VmtLink := Ofs(TypeOf(TSomeObject)^);
```

Die Felder *Load* und *Store* enthalten die Adressen der Methoden *Load* bzw. *Store* Ihres Objekts:

```
RSomeObject.Load := @TSomeObject.Load;
RSomeObject.Store := @TSomeObject.Store;
```

Um das Feld *Next* müssen Sie sich nicht kümmern, es wird von *RegisterType* zugewiesen. Dieses Feld vereinfacht den internen Gebrauch einer verketteten Liste von Stream-Registrierungs-Records.

Registrierung

Sobald Sie den Registrierungs-Record konstruiert haben, rufen Sie *RegisterType* mit dem Record als Parameter auf. Wenn Sie das neue *TMagritte*-Objekt für den Gebrauch mit Streams registrieren wollen, verfassen Sie folgenden Code:

```
const
  RMagritte: TStreamRec = (
    ObjType: 100;
    VmtLink: Ofs(TypeOf(TMagritte)^);
    Load: @Magritte.Load;
    Store: @Magritte.Store
  );
RegisterType(RMagritte);
```

Nun können Sie Instanzen des neuen Objekttyps mit der *Put*-Methode in jeden ObjectWindows-Stream einbringen und diese mit *Get* wieder von dort lesen.

Registrierung von Standardobjekten

ObjectWindows definiert Stream-Registrierungs-Records für alle Standardobjekte. Zusätzlich definiert die Unit *WObjects* eine *RegisterWObjects*-Prozedur, die alle Objekte dieser Unit registriert.

Der Stream-Mechanismus

Nachdem wir jetzt wissen, wie Streams verwendet werden, wollen wir noch kurz die Hintergründe beleuchten. Was passiert mit den Objekten, wenn sie mit einer *Get*- oder *Put*-Methode verarbeitet werden? Diese Frage soll in den folgenden Abschnitten beantwortet werden.

Die Put-Methode

Wird mit der *Put*-Methode ein Objekt zu einem Stream gesendet, sucht der Stream mit Hilfe des VMT-Zeigers vom Offset 0 des Objekts in seiner Liste der registrierten Typen nach einer Übereinstimmung. Wird eine Übereinstimmung gefunden, ermittelt der Stream die Registrierungs-ID des Objekts und schreibt sie in das Ziel des Streams. Anschließend ruft der Stream die *Store*-Methode des Objekts auf und beendet damit das Schreiben des Objekts. Die *Store*-Methode bedient sich der *Write*-Prozedur des Objekts, um die korrekte Anzahl von Bytes in das Ziel zu schreiben.

Das Objekt benötigt keine Kenntnisse über die Eigenschaften des Streams. Gleichgültig, um welche Art von Stream es sich handelt, sei es eine Datei, ein Teil des EMS-Speichers etc., das Objekt muß dem Stream nur mitteilen, daß es gespeichert werden will, die restliche Arbeit wird vom Stream erledigt.

Die Get-Methode

Wenn Sie mit der *Get*-Methode ein Objekt aus einem Stream lesen, wird zunächst die ID ermittelt und in der Liste der registrierten Typen nach einer Übereinstimmung gesucht. Wird eine Übereinstimmung gefunden, übergibt der Registrierungs-Record die Position der *Load*-Methode und der VMT des Objekts an den Stream. Anschließend wird die *Load*-Methode aufgerufen, um die korrekte Anzahl von Bytes aus dem Stream zu lesen.

Die *Get*-Methode weist den Stream an, sein nächstes Objekt an die Position zu schreiben, die durch den angegebenen neuen Zeiger bezeichnet wird. Für das Objekt ist es dabei gleichgültig, um welche Art von Stream es sich handelt. Der Stream liest die entsprechende Anzahl von Bytes mit der *Load*-Methode des Objekts, wobei diese wiederum auf der *Read*-Methode des Streams basiert.

Die Abläufe beim *Put*- und *Get*-Prozeß zeigen, wie wichtig es ist, einen Typ zu registrieren, bevor er für die Stream-Ein-/Ausgabe verwendet wird.

nil-Objektzeiger

Es ist möglich, ein nil-Objekt in einen Stream zu schreiben. In diesem Fall wird ein Word mit dem Wert 0 im Stream gespeichert. Beim Lesen einer ID mit dem Wert 0 liefert der Stream einen nil-Zeiger zurück. 0 ist daher reserviert und kann nicht als ID eines Stream-Objekts verwendet werden. ObjectWindows reserviert die Stream-IDs 0 bis 99 für die interne Verwendung.

Kollektionen in Streams: Ein Beispiel

Aus Kapitel 20 »Kollektionen« wissen wir, daß eine Kollektion verschiedene, aber miteinander verwandte Objekte enthalten kann. Die gleiche polymorphe Fähigkeit trifft auch für die Streams selbst zu, d.h. eine gesamte Kollektion kann in einem Stream auf Platte/Diskette gespeichert und von einem anderem Programm später wieder geladen werden.

Wenn wir uns das Programm COLLECT4.PAS unter diesem Aspekt ansehen, stellt sich folgende Frage: Wie muß das Programm verändert werden, damit die Kollektion in einen Stream eingebracht wird?

Die Antwort ist bemerkenswert einfach: Zunächst muß dem Basisobjekt *TGraphObject* »beigebracht« werden, wie es seine Daten (X und Y) in einem Stream speichern kann. Dafür kann die *Store*-Methode benutzt werden. Dann

wird für jeden abgeleiteten Typ von *TGraphObject*, der Felder hinzufügt, ebenfalls eine *Store*-Methode definiert (*TGraphPie* fügt zum Beispiel *ArcStart* und *ArcEnd* hinzu).

Anschließend wird für jeden Objekttyp, der tatsächlich gespeichert werden soll, ein Registrierungs-Record aufgebaut. Beim Start des Programms müssen diese Typen dann registriert werden. Danach sind nur noch normale Datei-Ein-/Ausgabeoperationen nötig: Deklarieren einer Stream-Variablen, Erzeugen eines neuen Streams, Speichern der gesamten Kollektion im Stream (dafür ist nur eine einfache Anweisung nötig) und Schließen des Streams.

Hinzufügen der Store-Methoden

Die *Store*-Methoden sind im folgenden aufgelistet. *PGraphEllipse* und *PGraphRect* benötigen keine *Store*-Methoden, da sie zu den von *PGraphObject* geerbten Feldern keine weiteren hinzufügen:

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    Rect: TRect;
    constructor Init(Bounds: TRect);
    procedure Draw(DC: HDC); virtual;
    procedure Store(var S: TStream); virtual;
  end;

  PGraphEllipse = ^TGraphEllipse;
  TGraphEllipse = object(TGraphObject)
    procedure Draw(DC: HDC); virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    procedure Draw(DC: HDC); virtual;
  end;

  PGraphPie = ^TGraphPie;
  TGraphPie = object(TGraphObject)
    ArcStart, ArcEnd: TPoint;
    constructor Init(Bounds: TRect);
    procedure Draw(DC: HDC); virtual;
    procedure Store(var S: TStream); virtual
  end;
```

Zur Implementierung der *Store*-Methode ruft jedes Objekt seine geerbte *Store*-Methode auf. Diese speichert alle geerbten Daten. Dann wird die *Write*-Methode des Streams aufgerufen, um die zusätzlichen Daten zu speichern:

```
procedure TGraphObject.Store(var S: TStream);
begin
  S.Write(Rect, SizeOf(Rect));
end;
```

```
procedure TGraphPie.Store(var S: TStream);
begin
  TGraphObject.Store(S);
  S.Write(ArcStart, SizeOf(ArcStart));
  S.Write(ArcEnd, SizeOf(ArcEnd));
end;
```

TGraphObject muß *TObject.Store* nicht aufrufen, da *TObject* über keine Daten verfügt, die gespeichert werden müssen.

Die *Write*-Methode von *TStream* schreibt die Daten im binären Format. Ihr erster Parameter kann eine Variable eines beliebigen Typs sein. *TStream.Write* hat aber keine Möglichkeit, festzustellen, aus wievielen Bytes die Variable besteht. Der zweite Parameter übergibt daher die Größe dieser Variable an *TStream.Write*. Verwenden Sie zur Angabe der Größe die normale *SizeOf*-Funktion. Der Compiler stellt dann sicher, daß immer die korrekte Anzahl von Bytes gelesen bzw. geschrieben wird.

Registrierungs-Records

Der letzte Schritt besteht aus der Registrierung einer Record-Konstanten für jeden abgeleiteten Typ. Hier sollten die Namenskonventionen von ObjectWindows verwendet werden, d.h. für den Typ wird anstelle des *T* ein *R* als erster Buchstabe verwendet.

Hinweis: Jeder Registrierungs-Record erhält eine eindeutige Objekt-ID (*ObjType*). ObjectWindows reserviert die IDs 0 bis 99 für seine Standardobjekte. Zur Vermeidung von Duplikaten sollten alle IDs an einem zentralen Ort verwaltet werden.

```
const
  RGraphEllipse: TStreamRec = (
    ObjType :150;
    VmtLink: Ofs(TypeOf(TGraphEllipse)^);
    Load: nil;                          { Noch keine Load-Methode }
    Store: @TGraphEllipse.Store);

  RGraphRect: TStreamRec = (
    ObjType :151;
    VmtLink: Ofs(TypeOf(TGraphRect)^);
    Load: nil;                          { Noch keine Load-Methode }
    Store: @TGraphRect.Store);

  RGraphPie: TStreamRec = (
    ObjType :152;
    VmtLink: Ofs(TypeOf(TGraphPie)^);
    Load: nil;                          { Noch keine Load-Methode }
    Store: @TGraphPie.Store);
```

Für *TGraphObject* wird kein Registrierungs-Record benötigt, da es sich hier um einen abstrakten Typ handelt, der nie instantiiert oder in einer Kollektion bzw. einem Stream verwendet wird.

Die *Load*-Zeiger der Registrierungs-Records werden hier auf nil gesetzt, da in diesem Beispiel nur Daten in einem Stream gespeichert werden. Im nächsten Beispiel (STREAM2.PAS) werden *Load*-Methoden definiert und die Registrierungs-Records aktualisiert.

Registrierung

Die Records müssen registriert werden, bevor mit ihnen Ein- und Ausgabeoperationen durchgeführt werden können. Am einfachsten ist es, sie in einer Prozedur zusammenzufassen und diese am Beginn des Programms (oder in der *Init*-Methode der Anwendung) aufzurufen:

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
  RegisterType(RGraphEllipse);
  RegisterType(RGraphRect);
  RegisterType(RGraphPie);
end;
```

Beachten Sie, daß jeder Objekttyp, den das Programm in einen Stream einbringt, von Ihnen registriert werden muß. Deshalb müssen Sie den Typ *TCollection* registrieren, obwohl Sie ihn nicht definiert haben. Die Registrierung erfolgt mit dem *RCollection*-Record (hier zeigt sich, wie sich die Programmierung durch die Einführung von Namenskonventionen vereinfachen läßt).

Schreiben in den Stream

Jetzt müssen Sie lediglich noch die üblichen Datei-Ein-/Ausgabeoperationen durchführen: Erzeugen eines Streams, Speichern der Daten (einer Kollektion) im Stream und Schließen des Streams. Für die Übertragung der Elemente der Kollektion müssen keine *ForEach*-Iteratoren geschrieben werden. Der Stream wird mit der *Put*-Methode angewiesen, die gesamte Kollektion zu speichern (das vollständige Listing ist in der Datei STREAM1.PAS enthalten).

```
var
  ...
  GraphicsStream: TBufStream;
begin
...
  StreamRegistration;                    { Objekte registieren }
  GraphicsStream.Init('GRAPH.STM', stCreate, 1024);
  GraphicsStream.Put(GraphicsList);      { Kollektion ausgeben }
  if GrapicsStream.Status <> 0 then
    Status := em_Stream;
  GraphicsStream.Done;                   { Stream schließen }
end;
```

Es wird eine Datei erzeugt, die alle Informationen enthält, die für das Einlesen der Kollektion in den Speicher erforderlich sind. Beim Öffnen des Streams und dem Laden der Kollektion (siehe STREAM2.PAS) werden alle verborgenen Verbindungen zwischen der Kollektion und ihren Elementen und zwischen den

Objekten und ihren virtuellen Methodentabellen wiederhergestellt. Der folgende Abschnitt erläutert, wie Objekte, die Verbindungen zu anderen Objekten enthalten, in einen Stream eingebracht werden können.

Wer speichert was?

Folgender Aspekt ist bei Streams unbedingt zu beachten: Nur der Besitzer eines Objekts sollte das Objekt auch in einen Stream schreiben. Darauf müssen Sie auch im traditionellen Pascal achten: Der Besitzer eines Zeigers ist auch derjenige, der den Zeiger freigeben sollte.

In realen Anwendungen haben viele Objekte einen Zeiger auf eine bestimmte Struktur. Wenn es dann zu Stream-Ein-/Ausgaben kommt, müssen Sie entscheiden, wer die Struktur »besitzt«. Denn nur der Besitzer sollte die Struktur in den Stream einbringen. Andernfalls stehen im Stream mehrere Kopien der nur einmal vorhandenen Struktur. Wenn Sie dann den Stream lesen, werden mehrere Instanzen der Struktur erzeugt. Dabei zeigt jedes der ursprünglichen Objekte auf seine eigene Kopie und nicht auf die ursprüngliche Struktur.

Felder in Streams

Es gibt Situationen, in denen Zeiger auf eine Gruppe von untergeordneten Fenstern günstigerweise in lokalen Variablen einer Instanz (den Datenfeldern des Objekts) gespeichert werden sollten. So könnte zum Beispiel ein Dialogfenster Zeiger auf seine Steuerelementobjekte in Feldern mit (den Inhalt beschreibenden) Namen speichern (z.B. *OKButton* oder *FileInputLine*) und damit den Zugriff erleichtern. Wenn das untergeordnete Fenster erzeugt wird, verfügt das übergeordnete Fenster über *zwei* Zeiger dieses Fenster. Wenn Sie dies nicht berücksichtigen, werden beim Lesen des Objekts aus dem Stream doppelte Instanzen erzeugt.

Die Lösung bringen die *TWindowsObject*-Methoden *GetChildPtr* und *PutChildPtr*. Beim Speichern eines Feldes, das auch ein untergeordnetes Fenster ist, wird nicht – wie bei anderen Variablen – der Zeiger gespeichert. In diesem Falle wird die Methode *PutChildPtr* aufgerufen, die eine Referenz auf die ordinale Position des untergeordneten Fensters in der Liste dieser Gruppe speichert. Wenn Sie dann die Gruppe mit *Load* aus dem Stream lesen, können Sie die Methode *GetChildPtr* aufrufen. Diese Methode gewährleistet, daß das Feld und die Liste der untergeordneten Fenster auf dasselbe Objekt zeigen.

Im folgenden sehen Sie ein Beispiel für den Gebrauch von *GetChildPtr* und *PutChildPtr* in einem einfachen Fenster:

```
type
  TDemoWindow = object(TWindow)
  Msg: PStatic;
```

```
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
  end;

  constructor TDemoWindow.Load(var S: TStream);
  begin
    TWindow.Load(S);
    GetChildPtr(S, Msg);
  end;

  procedure TDemoWindow.Store(var S: TStream);
  begin
    TWindow.Store(S);
    PutChildPtr(S, Msg);
  end;
```

Sehen wir uns nun näher an, wodurch sich die *Store*-Methode von dem normalen *Store* unterscheidet. Nachdem das Fenster regulär gespeichert wurde, müssen Sie nur eine Referenz auf das Feld *Msg* speichern und nicht das Feld selber, wie das üblicherweise der Fall ist. Das Schalter-Objekt wird durch den Aufruf von *TWindow.Store* als untergeordnetes Fenster gespeichert. Sie müssen jetzt nur den Stream darüber informieren, daß *Msg* auf dieses untergeordnete Fenster zeigt. Die *Load*-Methode führt dieselben Schritte in umgekehrter Richtung aus: Zuerst wird das Fenster und sein untergeordnetes Schalter-Fenster geladen. Danach wird in *Msg* der Zeiger auf das untergeordnete Fenster gespeichert.

Instanzen von nebengeordneten Fenstern

Eine ähnliche Situation kann entstehen, wenn ein Fenster ein Feld beinhaltet, das auf ein nebengeordnetes Fenster zeigt. Ein Fenster wird als *nebengeordnet* bezeichnet, wenn es zusammen mit einem anderen Fenster zu demselben übergeordneten Fenster gehört. Stellen Sie sich zum Beispiel vor, daß Sie ein Editierfeld und zwei Schaltfelder haben, die festlegen, ob das Editierfeld aktiviert werden kann oder nicht. Wenn sich der Status eines Schaltfeldes ändert, wird das Editierfeld entsprechend aktiviert oder deaktiviert. Da *TActivateRadioButton* von einem Editierfeld, das ebenfalls ein Element desselben Fensters ist, wissen muß, muß es als Instanzenvariable hinzugefügt werden.

Ebenso wie bei untergeordneten Fenstern kann es auch bei nebengeordneten Fenstern zu Problemen kommen, wenn deren Referenzen in Streams geschrieben oder von dort gelesen werden. Die Schwierigkeiten werden auf ähnliche Weise wie bei den untergeordneten Fenstern gelöst: die TWindowsObject- Methoden PutSiblingPtr und GetSiblingPtr greifen auf nebengeordnete Fenster zu:

```
  type
    TActivateRadioButton = object(TRadioButton)
    EditControl: PEdit;
    ...
```

```
    constructor Load(var S: TStream);
    procedure Store(var S: Tstream); virtual;
    ...
  end;

  constructor TActivateRadioButton.Load(var S: TStream);
  begin
    TRadioButton.Load(S);
    GetSiblingPtr(S, EditControl);
  end;

  procedure TActivateRadioButton.Store(var S: TStream); virtual;
  begin
    TRadioButton.Load(S);
    PutSiblingPtr(S, EditControl);
  end;
```

Zusätzlich zu beachten ist nur, daß Referenzen auf nebengeordnete Fenster geladen werden können, die selbst noch nicht geladen sind (d.h. diese Fenster folgen später in der Liste der untergeordneten Fenster und stehen deshalb auch weiter hinten im Stream). ObjectWindows verwaltet dies selbständig, indem es derartige Vorwärts-Referenzen überwacht und erst dann auflöst, wenn alle untergeordneten Fenster geladen sind. Beachten Sie daher, daß nebengeordnete Referenzen erst gültig werden, wenn die *Load*-Prozedur vollständig ausgeführt ist. Aus diesem Grund sollten Sie keinen Code in die *Load*-Methoden schreiben, der untergeordnete Fenster benutzt, die von ihren nebengeordneten Fenstern abhängig sind. Vorsicht: Die Ergebnisse sind nicht vorhersehbar.

Streams kopieren

TStream verfügt über die Methode *CopyFrom(S,Count)*, die *Count* Bytes aus dem gegebenen Stream *S* kopiert. Mit *CopyFrom* kann der gesamte Inhalt eines Streams in einen anderen kopiert werden. Wenn Sie wiederholt auf einen Stream auf der Platte zugreifen, könnten Sie diesen beispielsweise in einen EMS-Stream kopieren, um die Zugriffsgeschwindigkeit zu erhöhen:

```
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

Streams mit Direktzugriff

Bis hierher haben wir Streams als sequentielle Dateien behandelt: Sie fügen mit *Put* Objekte an das Ende des Streams ein und lesen Sie mit *Get* in derselben

Reihenfolge. ObjectWindows kann aber noch mehr: Sie können einen Stream als virtuelle Datei mit Direktzugriff behandeln. Zusätzlich zu *Get* und *Put*, die *Read* und *Write* für Dateien entsprechen, verfügen Streams über Methoden, die den Funktionen *Seek*, *FilePos*, *FileSize* und *Truncate* entsprechen.

- Die Stream-Prozedur *Seek* verschiebt den aktuellen Stream-Zeiger auf eine angegebene Position (in Bytes vom Anfang des Streams an gerechnet) – wie dies die Standardprozedur *Seek* von Pascal auch ausführt.
- Die Funktion *GetPos* ist die Umkehrung der *Seek*-Prozedur. Sie liefert einen Longint-Wert mit der aktuellen Stream-Position zurück.
- *Die Funktion GetSize* liefert die Größe des Streams in Bytes zurück.
- Die Prozedur *Truncate* löscht alle Daten ab der aktuellen Position im Stream und macht sie so zum Ende des Streams.

Für Streams mit Direktzugriff müssen Sie allerdings einen Index mitführen, der die Startposition jedes Objekts vermerkt. Verwenden Sie für den Index eine Kollektion.

Nicht-Objekte in Streams

Sie können in einen Stream auch Elemente einbringen, die keine Objekte sind. Dazu müssen Sie aber etwas anders vorgehen. Normalerweise wird bei der *Get*- und *Put*-Methode ein von *TObject* abgeleitetes Objekt geladen bzw. gespeichert. Soll ein Stream erzeugt werden, der nicht aus Objekten sondern aus anderen Elementen besteht, müssen Sie mit den Prozeduren *Read* und *Write* eine angegebene Anzahl von Bytes aus dem Stream lesen oder in ihn schreiben. Es handelt sich dabei um den gleichen Mechanismus, mit dem die *Get*- und *Put*-Methode Daten für die Objekte liest bzw. schreibt. Bei dieser Vorgehensweise wird aber der von den Methoden *Get* und *Put* verwendete VMT-Mechanismus umgangen.

Der Entwurf eigener Streams

Dieser Abschnitt gibt einen Überblick über Methoden und Fehlerbehandlung der ObjectWindows-Streams. Sie können sie bei der Entwicklung neuer Stream-Arten verwenden.

TStream ist ein abstraktes Objekt, das erweitert werden muß, um einen brauchbaren Stream-Typ zu erhalten. Die meisten Methoden von *TStream* sind abstrakt und müssen in den abgeleiteten Typen implementiert werden. Einige Methoden sind von den abstrakten *TStream*-Methoden abhängig. Voll implementiert sind nur die *TStream*-Methoden *Error*, *Get* und *Put*. *GetPos*, *GetSize*, *Read*, *Seek*, *SetPos*, *Truncate* und *Write* müssen überschrieben werden. Verfügt der

abgeleitete Objekttyp über einen Puffer, sollte auch die *Flush*-Methode überschrieben werden.

Fehlerbehandlung in Streams

TStream verfügt über eine Methode namens *Error(Code, Info)*, die aufgerufen wird, wenn der Stream auf einen Fehler trifft. *Error* setzt lediglich das *Status*-Feld des Streams auf eine der Konstanten, die in Kapitel 6 »Außerhalb der Objekte« im Abschnitt »stXXX-Konstanten« des *Referenzhandbuchs* aufgelistet sind.

Das Feld *ErrorInfo* ist undefiniert, außer *Status* ist auf *stGetError* oder *stPutError* gesetzt. Ist *Status stGetError*, enthält das Feld *ErrorInfo* die Stream-ID-Nummer des unregistrierten Typs. Ist *Status stPutError*, enthält das *ErrorInfo*-Feld den VMT-Offset des Typs, der in den Stream eingebracht werden sollte. *TStream.Error* kann für die Fehlerbehandlung (auch von Laufzeitfehlern) überschrieben werden.

Index

$M-Compiler-Direktive 186
$R Compiler-Direktive 44, 47, 246f.
$X+ Compiler-Direktive 45
&-Zeichen 148

A
Abbildungsmodi 221
Accelerator-Tasten
  Befehle und 253
  laden 91, 253
  MDI und Botschaften 92
AddString
  TListBox-Methode 144
AddString-Funktion 256
Aktionsschalter
  Beispiel 153
  Botschaften und 150
  konstruieren 150
  Stile 149
Andauernder Datentransfer 215
Antwortmethoden 81
Anwendungen 7
  »Hello, World« 16
  Anforderungen 8
  beenden 26, 88
  Botschaftsschlangen 81
  Botschaftsschleifen 81
  Destruktor 21, 88
  globale Variable 267
  Hauptprogramm 88
  Initialisierung 88ff.
  Konstruktor 21, 88
  Name 21
  Objekte 20, 73, 87ff.
  schließen 93
  sichere 266
  Start 17
  Struktur 15
  Voraussetzungen 20
Anwendungsobjekt 268
API
  Windows 77
Application
  globale Variable 88, 267
Arc-Funktion 235
Attr
  TWindow-Feld 108
Attribute
  erstellen 111
  registrieren 111, 254
  Registrierungs- 113, 254
  siehe auch Registrierungsattribute
Auswahlfelder
  Beispiel 153
  Transferpuffer 170
AutoMode
  TScroller-Feld 121

B
Befehle
  Accelerator-Tasten und 253
  antworten auf 46
  Menü- 160
Benachrichtigungsbotschaften
  Bildlaufleisten und 158
Bereich
  Fenter rollen 122
  Bildlaufleisten 156
bf_Checked-Konstante 152
bf_Grayed-Konstante 152
bf_Unchecked-Konstante 152
Bibliotheken
  siehe DLLs
Bildlaufleisten 154ff.
  ändern 156
  automatisches Rollen und 121
  Beispiel 158
  Bereich 155

Botschaften und 156
Informationen über 155
konstruieren 154
Position 155
rollen von Fenstern und 119
Seitengröße 155
Stile 154
Transferpuffer 170
Zeilengröße 155
Bildschirm löschen 32, 47
Bildschirmkontext 29, 219ff.
Bitmaps und 220
Einheiten 221
Funktionen 29
Handle 29
Hintergrundfarbe 222
Koordinaten 221
verwalten 220
Zeichenwerkzeuge und 222ff.
Bitmaps
als Menüpunkte 257, 259
als Pinsel 257
Bildschirmkontext und 220
geräteunabhängige 221
Kompatibilität 220
laden 256
löschen 257
vordefinierte 256
Botschaften 14, 80
Aktionsschalter 150
Antwortmethoden 101
anwenderdefinierte 86
beantworten 25, 100
bearbeiten 88, 92, 99
Befehls- 46, 85, 102
Benachrichtigungs- 85, 129, 141
Bereiche 85
Bildlaufleisten und 156
Editierfelder und 161
Ereignisse und 100
Gruppen und 153
ID des untergeordneten Fensters 65, 85, 102, 129, 150
Listen-Benachrichtungsbotschaften 145
Menüs und 45
Multidokumentschnittstelle 178
Parameter 31, 82, 101
Records 31
senden 84
Standardantworten 84, 104
Steuerelemente und 84f., 128, 140
Steuerelementobjekte und 131
Typen 83
Übermittlung zwischen Anwendungen 207
verteilen 81
Botschaftsantwort-Methoden 81
Botschaftsschlangen 81
Botschaftsschleifen 81, 92
bs_AutoCheckBox-Stil 151
bs_AutoRadioButton-Stil 151
bs_DefPushButton-Stil 149
bs_DIBPattern-Stil 225
bs_Hatched-Stil 225
bs_Hollow-Stil 225
bs_Pattern-Stil 225
bs_PushButton-Stil 149

C

Callback-Funktionen 79
Cancel
TDialog-Methode 127
CanClose
Dialogfenter und 127
TApplication-Methode 26, 93
TWindowsObject-Methode 26, 55, 93
untergeordnete Fenster und 26
cbs_AutoHScroll-Stil 167
cbs_Sort-Stil 167
cf_Bitmap-Konstante 203
cf_MetafilePict-Konstante 203
cf_Sylk-Konstante 203
cf_Text-Konstante 203f.
cf_Tiff-Konstante 203
Check
TCheckBox-Methode 152
ChildList
TMDIWindow-Feld 176
TObjectWindows-Feld 98
ChildMenuPos
TMDIWindow-Feld 177
Chord-Funktion 237
Clear
TEdit-Methode 163
TStatic-Methode 149
ClearList
TListBox-Methode 144
ClearModify
TEdit-Methode 163
Client-Bereich 10
rollen 118f.
Client-Fenster 208
siehe auch MDI

ClientWnd
  TMDIWindow-Feld 176
Clipping-Regionen 221
CloseClipboard-Funktion 204f.
cm_First-Konstante 45, 102, 253
CMEditCopy
  TEdit-Methode 160
CMEditUndo
  TEdit-Methode 160
CMTileChildren
  TMDIWindow-Methode 179
Common User Access (CUA) 261
Compiler-Direktiven
  $M 186
  $R 44, 47, 246f.
  $X+ 45
CopyFrom
  TStream-Methode 299
Create
  Aufruf durch andere Methoden 111
  MakeWindow und 111
  MakeWindow vs. 52, 267
  TDialog-Methode 38
  TWindowsObject-Methode 52, 96
CreateChild
  TMDIWindow-Methode 177
CreatePen-Funktion 38, 223
CreatePenIndirect-Funktion 223
cs_ Konstanten 114

D
Datei-Dialogfenster 135
  Konstruktion 135
  Ressourcen und 135
Dateidialoge
  verwenden 136
Dateien
  Header- 250
  Objekt-Ein-/Ausgabe 285
  Objekte und 285
  öffnen 53, 135
  .RC 247
  .RES 246
  Ressourcen-Makro 247
  schließen 53
  sichern 135
  Textdateien editieren 115
  Typenprüfung und 286
  virtuelle Methodentabellen und 286
Dateifenster 115, 117
Datenkonvertierung
  zeitverzögerte 207
Datenstrukturen 78
DDE
  siehe dynamischer Datenaustausch
DefWindowProc-Funktion
  TWindowsObject-Methode 84
DefWndProc
  TControl-Methode 131
DeleteLine
  TEdit-Methode 164
DeleteObject-Funktion 257
DeleteSelection
  TEdit-Methode 164
DeleteString
  TListBox-Methode 144
DeleteSubText
  TEdit-Methode 164
DeltaPos
  TScrollBar-Methode 156
Destroy
  TWindowsObject-Methode 96
Dialogfenster 50, 125ff.
  als untergeordnete Fenster 125
  anzeigen 54
  ausführen 126
  Datei- 52, 54
  Daten ermitteln 127
  Fenster vs. 125
  konstruieren 53
  laden 254
  modale und nicht-modale 126ff.
  nicht-modale 92
  -Objekte 74, 125ff.
  Pop-up-Fenster und 52
  Ressourcen und 52, 125, 254
  Rückgabewerte 52
  schließen 127
  Standard- 265
  Steuerelemente und 128
  Stringtabellen und 255
  Transfermechanismus 170
  verwalten 127
  vordefiniertes 52
DisableAutoCreate
  TWindowsObject-Methode 98
DisableTransfer
  TControl-Methode 171
DLLs 11, 197
  Beispiel 198
  Units und 199
  Zugriff auf 197
Done
  TApplication-Methode 21, 88, 93

DOS 16
Dynamische Linkbibliotheken
  siehe DLLs
Dynamische Methoden 45
  Botschaften und 86
Dynamischer Datenaustausch (DDE) 12, 203, 208
  andauernder Datentransfer 215
  Makrobefehle 217
  Methoden 211

E
Editierfelder 158
  ändern 163
  Beispiel 164
  Botschaften und 161
  Editierfenster und 115
  Informationen über 161
  konstruieren 159
  löschen 163
  mehrzeiliges Editieren 158
  Stile 159
  Transferpuffer 170
  Zwischenablage und 160
Editierfenster 115
  Dateifenster und 117
Editor
  TFileWindow-Feld 117
Eingabe
  Überprüfung 127
Eingabedialoge 134
  Beispiel 134
  Konstruktion 134
  Ressourcen und 134
Eingabefelder 36, 38
Ellipse-Funktion 237
em_InvalidChild-Konstante 269
em_InvalidClient-Konstante 269
em_InvalidMainWindow-Konstante 269
em_InvalidWindow-Konstante 269
em_OutOfMemory-Konstante 269
EmptyClipboard-Funktion 204
EnableAutoCreate
  TWindowsObjectMethode 98
EnableKBHandler
  TWindowsObject-Methode 143
EnableTransfer
  TControl-Methode 171
EndDlg
  TDialog-Methode 127
Ereignisgesteuerte Architektur 9
Ereignisgesteuerte Programmierung 100
Ereignisse 65
  Botschaften und 100
  Steuerelemente und 65
Error
  TCollection-Methode 283
  TStream-Methode 289, 300
ErrorInfo
  TStream-Feld 289, 301
es_AutoHScroll-Stil 159
es_AutoVScroll-Stil 159
es_Left-Stil 159
es_Multiline-Stil 159
es_UpperCase-Stil 165
ExecDialog
  Error und 268
  Execute vs. 55, 268
  Rückgabewerte 268
  TApplication-Methode 55
Execute
  ExecDialog vs. 55, 268
externe Direktive 197

F
Farbe
  Hintergrund 112
Farben 221, 224, 239ff.
  Abwandlung 221
  Bildschirmkontext und 219, 222
  Hintergrund- 114, 219, 222
  Intensität 223
  setzen 223
  Stifte 223
  verwenden 266
Farbpaletten
  siehe Paletten
Fehler
  Streams 289, 301
Fenster
  anzeigen 59
  Attribute erstellen 111
  Attribute, voreingestellte 109
  Bildschirmkontext und 229
  Grafikanzeige in 229
  Grafikfunktionen aufrufen 230
  Hintergrundfarbe 114
  Konstruktor 44
  Koordinaten 30
  Menüattribute 44
  Neuzeichnen 229f.
  rollen 117
  schließen 66
  Sicherheitsbereich anlegen 267

Standard- 77
Steuerelement initialisieren 171
Steuerelemente in 62
Titel 109
Transfermechanismus 170
verändern 265
zeichnen in 229
Fensterelemente
erstellen 110
Fensterobjekte 107
Attribute 108
initialisieren 108
Konstruktor 109
FirstThat
TCollection-Methode 276
TWindowsObject-Methode 99
Flags
globaler Speicher 190
lokaler Speicher 187
ForEach
TCollection-Methode 275
TWindowsObjectMethode 99
Formate konvertieren 206
FreeItem
TCollection-Methode 273
Funktionen
API- 79
Callback- 79
Windows- 77

G

GDI.EXE 15
gemeinsame Code-Nutzung 197
geometrische Formen 236
Gerätetreiber
Bildschirmkontext und 219
Get
TStream-Methode 288, 293
GetCheck
TCheckBox-Methode 152
GetChildPtr 297
GetChildWithID
TWindow-Methode 141
GetClassName
TDlgWindow-Methode 133
überladen 112
GetClipboardData-Funktion 205
GetCount
TListBox-Methode 145
GetItem
TCollection-Methode 273
GetItemHandle
TDialog-Methode 129
GetLine
TEdit-Methode 161f.
GetPos
TStream-Methode 300
GetPosition
TScrollBar-Methode 155
GetRange
TScrollBar-Methode 155
GetSelection
TEdit-Methode 163
GetSelIndex
TListBox-Methode 147
GetSelString
TListBox-Methode 147
GetSize
TStream-Methode 300
GetStockObject-Funktion 223
GetString
TListBox-Methode 145
GetStringLen
TListBox-Methode 145
GetSubText
TEdit-Methode 163
GetText
TEdit-Methode 161f.
GetWindowClass
Mauszeiger und 254
Symbole und 254
TDlgWindow-Methode 133
TWindow-Methode 113
überladen 112
GetWindowClass-Methode 255
GlobalAlloc-Funktion 190
Rückgabewerte 191
GlobalCompact-Funktion 194
GlobalDiscard-Funktion 192
Globaler Heap 190
Globaler Speicher 185, 190
64-KByte-Grenze 193
abfragen 193
fixieren 192
Flags 190
freigeben 192
Größe 193
reservieren 190
Sperre aufheben 192
sperren 192
Sperrenzähler 192
ungültiges Handle 191
verwerfen 194
zusammenfassen 194

GlobalFix-Funktion 192
GlobalFlags-Funktion 193
GlobalFree-Funktion 192
GlobalLock-Funktion 191
GlobalLRUNewest-Funktion 194
GlobalLRUOldest-Funktion 194
GlobalReAlloc-Funktion 193
GlobalSize-Funktion 193
GlobalUnix-Funktion 192
GlobalUnlock-Funktion 191
GlobalUnwire-Funktion 192
GlobalWire-Funktion 192
gmem_DDEShare-Flag 194, 208
gmem_Discardable-Flag 191, 193
gmem_Fixed-Flag 191, 193
gmem_Modify-Flag 193
gmem_Moveable-Flag 191, 193
gmem_NoCompact-Flag 191, 193
gmem_NoDiscard-Flag 191, 193
gmem_Not_Banked-Flag 194
gmem_ZeroInit-Flag 193
Grafische Geräteschnittstelle (GDI) 219
  Zeichenfunktionen 233
Gruppen
  Botschaften 153
  konstruieren 152

H

HAccTable
  TApplication-Feld 253
Handles
  Bildschirmkontext 29
  Fenster 23, 96
  Speicher 10
  Speicherverwaltung und 185
  Steuerelemente 129
  Zeichenwerkzeuge 36
Hauptfenster 89
  Anwendungsobjekte und 87
  Bedingungen 18
  erstellen 20
  Größe 109, 265
  Initialisierung 89
  Menü und 44
  Schließen 90
  Titel 89, 91
hbrBackground
  TWndClass-Feld 114
hCursor
  TWndClass-Feld 114
Header-Dateien 250
hIcon
  TWndClass-Feld 114
HideList
  TComboBox-Methode 168
Hilfesystem
  Windows 265
Hintergrundfarbe
  Bildschirmkontext 222
  Fenster 114
Hot Link 215
HWindow
  TWindowsObject-Feld 23

I

ID-Nummern
  von Objekten 291
  von Streams 293
id_First-Konstante 65, 103, 153
id_OK-Konstante 171
idc_Arrow-Konstante 114
idc_Beam-Konstante 113
idc_Cross-Konstante 114
idc_IBeam-Konstante 114
idc_Wait-Konstante 114
IDE
  s. Integrierte Entwicklungsumgebung
idi_ Konstanten 255
idi_Application-Konstante 114
IDs
  Dialogressource 125
  Ressourcen 43
  Steuerelemente 64
  Stringtabellen 255
  untergeordnete Fenster 103
  von Menüauswahlen 253
Init
  InitResource vs. 171
  TApplication-Methode 21, 88
  TButton-Methode 150
  TCheckBox-Methode 151
  TComboBox-Methode 167
  TControl-Methode 139, 170
  TDialog-Methode 126
  TEdit-Methode 159
  TFileDialog-Methode 135
  TFileWindow-Methode 117
  TGroupBox-Methode 152
  TInputDialog-Methode 134
  TListBox-Methode 143
  TMDIWindow-Methode 177
  TRadioButton-Methode 151
  TScroller-Methode 119, 154

TStatic-Methode 148
TWindow-Methode 45, 109
TWindowsObject-Methode 96
InitApplication
TApplication-Methode 88, 91
InitChild
TMDIWindow-Methode 177
InitClientWindow
TMDIWindow-Methode 177
InitInstance
TApplication-Methode 88, 90, 253
InitMainWindow
MDI und 177
TApplication-Methode 21, 88
InitResource
Init vs. 131
MakeWindows vs. 131
TControl-Methode 170
InitRessource
TApplication-Methode 169
Insert
TEdit-Methode 164
InsertString
TListBox-Methode 144
Instanzenverbindung 21
Integrierte Entwicklungsumgebung 2
InvalidateRect-Prozedur 32
IsClipboardFormatAvailable-
Funktion 205
IsModified
TEdit-Methode 163
IsVisibleRect
TScroller-Methode 124
Iterator-Methoden
untergeordnete Fenster 98
Iteratormethoden
FirstThat 276
ForEach 275
Kollektionen und 275ff.
LastThat 276

K

KERNEL.EXE 15
Klassen
Attribute 254
Fenster- 112ff.
Registrierung 112
Stile 114
Kollektionen 271ff.
Arrays 271
Beispiele 273ff.
dynamisch vergrößern 271
Fehler 282
Iteratormethoden und 275ff.
maximale Größe 282
Nicht-Objekte und 272
Polymorphie und 272, 280
sortierte 277
Speicherverwaltung 282
String- 278
Typenprüfung und 272
Zeiger und 272
Kombinationsfenster
ändern 168
aufspringende 165
aufspringende Listen- 166
Beispiel 168
einfache 165
konstruieren 167
Listen vs. 165
Listenbereich 168
Stile 167
Transferpuffer 169
Kommunikation
siehe dynamischer Datenaustausch
Konstanten
Befehls-IDs 102
IDs 46
Stil- 79
Konversation 208ff.
aufbauen 209
beenden 211
Koordinatensystem
Abbildungsmodus und 221
Bildschirmkontext und 221
von Windows 30

L

LastThat
TCollection-Methode 276
lbColor
TLogBrush-Feld 225
lbn_SelChange-Botschaften 146
lbStyle
TLogBrush-Feld 224
LineLength
TEdit-Methode 162
LineSize
TScrollBar-Feld 155
LineTo-Funktion 35, 234
Linien zeichnen 222, 234
Linienstile 223
Liste mit untergeordneten
Fenstern 97, 139

Listen 143ff.
abfragen 145
ändern 144
Auswahl von 147
Benachrichtigungsbotschaften 145
Konstruktion 143
Strings und 144ff.
Transferpuffer 170
lmem_Discardable-Flag 187, 189
lmem_Fixed-Flag 187
lmem_Modify-Flag 189
lmem_Moveable-Flag 187, 189
lmem_NoCompact-Flag 187, 189
lmem_NoDiscard-Flag 187, 189
lmem_ZeroInit-Flag 189
Load
TStreamRec-Feld 291
Load-Methode 289, 293
LoadAccelerator-Funktion 254
LoadMenu-Funktion 45, 253
LoadString-Funktion 256
LocalAlloc-Funktion 187
LocalCompact-Funktion 190
LocalDiscard-Funktion 188
localFlags-Funktion 189
LocalFree-Funktion 188
LocalLock-Funktion 187
LocalReAlloc-Funktion 189
LocalSize-Funktion 187, 189
LocalUnlock-Funktion 187
lokaler Heap 185ff.
Lokaler Speicher 185ff.
fester 187
Flags 187, 189
freigeben 188
Handles und 187
Maximalgröße 186f.
reservieren 187
verwerfbarer 188f.
lopnColor
TLogPen-Feld 223
lopnStyle
TLogPen-Feld 223
lopnWidth
TLogPen-Feld 223
LowMemory-Funktion 267
lParam
Botschaftsparameter 101
lpszMenuName
TWndClass-Feld 115
LRU
siehe Speicherverwaltung

M
MainWindow
TApplication-Methode 21
MakeWindow
Create und 111
Create vs. 52, 267
Error und 268
nicht-modaler Dialog und 126
Rückgabewerte 52
TApplication-Methode 52, 267
TWindowObject-Methode 111
Makrodatei 248
Mausbewegungen 32
Mauszeiger
Symbole vs. 254
MaxCollectionSize-Variable 282
MemAlloc-Funktion 183
Menu
TWindowAttr-Feld 44
Menüs 43
Befehle beantworten 253
Bitmaps und 259
Botschaften und 45
Hauptfenster 44
Laden 252
Ressourcen 43
Standard 264
MessageLoop
TApplication-Methode 93
Methoden
Botschaftsbeantwortung 81, 102
MoveTo-Funktion 35, 234
Multidokumentschnittstelle
(MDI) 12, 175
Beispiel 180
Botschaften und 178
Client-Fenster 175
Fenster konstruieren 176
Komponenten 175
Menüs und 179
Objekte 75, 176

N
Namenskonventionen 71
Nebengeordnete Fenster 298
New
TFileWindow-Methode 117
Next
TStreamRec-Feld 291
nf_First-Konstante 142
Nicht-Objekte
Kollektionen und 272

nil-Objekte
  Streams und 293
NumLines
  TEdit-Methode 162

O
ObjectWindows 13
  Hierarchie 71
  Konventionen 71
  Ressourcen 77
  Units 76
Objekte
  Anwendungen 20, 73, 87ff.
  Basis- 73
  Dialogfenster- 74
  Fenster- 23
  ID-Nummern 291
  lesen aus Streams 288
  Multidokumentschnittstelle 75
  neue ableiten 289
  nil 293
  Schnittstellen- 14, 73
  schreiben in Streams 288
  Steuerelement- 74
  Streamregistrierung 287
Objektfelder
  untergeordnete Fenster und 63
obm_Close-Bitmap 257
obm_DnArrow-Bitmap 257
obm_Zoom-Bitmap 257
OK
  TDialog-Methode 127
Open
  TFileWindow-Methode 117
OpenClipboard-Funktion 204f.

P
PageSize
  TScrollBar-Feld 155
Paint
  Bildschirmkontext und 40
  TWindow-Methode 40, 42, 230ff.
Paletten 221, 239ff.
  -Botschaften 242
  ändern 241
  erstellen 240
  Informationen über 241
  zeichnen mit 240
Parent
  TWindowsObject-Feld 24
PChar-Typ 45
Pie-Funktion 237
Pinsel
  Arten 222
  Bildschirmkontext und 222
  Bitmaps als 225, 257
  Farben 225
  Schraffurmuster 225
  virtuelle 224
Polygon-Funktion 238
PolyLine-Funktion 234
PostMessage-Funktion 209, 211
PrintWord-Prozedur 279
ProcessAccels
  TApplication-Methode 92
ProcessDlgMessage
  TApplication-Methode 92
ProcessMDIAccels
  TApplication-Methode 92
Put
  TStream-Methode 288, 292
PutChildPtr 297
  Beispiel 297
PutItem
  TCollection-Methode 273

R
Rahmenfenster
  MDI 175
.RC-Dateien 247
Read
  TStream-Methode 300
Read-Methode 293
Rectangle-Funktion 236
RegisterClipboardFormat-
  Funktion 203
RegisterType-Prozedur 290
Registrierung 296
  Beispiel 292
  Fenster 112
  Fensterklassen 111
  neue Typen und 290
  Records 291
  Streams 290
Registrierungs-Records 295
Registrierungsattribute
  ändern 112
  Hintergrundfarbe 114
  Standardmenü 115
  Stile 114
  Symbol 114
  Zeiger 114
ReleaseCapture-Funktion 35
.RES-Dateien 43, 246

Ressourcen 11, 245
  Bitmaps 256
  Dateidialoge und 136
  Dialogfenster und 125, 254
  Eingabedialoge und 134
  entfernen 245
  erstellen 246
  Header-Dateien und 250
  hinzufügen 246
  ID 43
  Laden 252
  Mauszeiger 254
  Menüs 43, 254
  Namen 253
  StdDlgs-Unit 135
  Steuerelemente und 137
  Strings 255
  verknüpfen 47
Ressourcen-Compiler 247
  Befehlszeilenoptionen 247
  Makrodateien 247f.
Result
  Message-Feld 101
RGB-Funktion 224
Rollen von Fenstern 117
  automatisches 121
  Beispiel 120
  Bereich ändern 122
  Bildlaufleisten und 119
  Einheiten ändern 122
  Position ändern 122
  Seitengröße ändern 123
  Zeichenmethoden und 123
RoundRect-Funktion 236
Run
  TApplication-Methode 88, 92

S

Save
  TFileWindow-Methode 117
SaveAs
  TFileWindow-Methode 117
SBBottom
  TScrollBar-Methode 158
SBLineDown
  TScrollBar-Methode 158
SBLineUp
  TScrollBar-Methode 158
SBPageDown
  TScrollBar-Methode 158
SBPageUp
  TScrollBar-Methode 158
sbs_Horz-Stil 154
sbs_TopAlign-Stil 154
sbs_Vert-Stil 154
SBThumbPosition
  TScrollBar-Methode 158
SBThumbTrack
  TScrollBar-Methode 158
SBTop
  TScrollBar-Methode 158
Schaltfelder
  Beispiel 153
Schnittstellenelemente 23
  anzeigen 97
  entfernen und freigeben 96
  erstellen 52, 96
  Fehler 268
Schnittstellenobjekte 14, 23, 73, 95ff.
  entfernen 97
  Fenster 108
  Konstruktor 96
  Steuerelemente 128
  Verknüpfung mit Steuer-
  elementen 131
  Zweck 96
Schraffurmuster 225
Schrift
  Bildschirmkontext und 219
  virtuelle 225
Scroll
  TEdit-Methode 164
ScrollBarTransferRec-Record 170
ScrollBy
  TScroller-Methode 122
Scroller-Attribute 118
ScrollTo
  TScroller-Methode 122
Seek
  TStream-Methode 300
SelectRange
  TEdit-Methode 164
SendDlgItemMsg
  TDialog-Methode 129
SendDlgItemsMsg
  TDialog-Methode 85
SendMessage-Funktion 85, 209f.
Server 208
SetCapture-Funktion 35
SetCheck
  TCheckBox-Methode 152
SetClipboardData-Funktion 204
SetPageSize
  TScroller-Methode 123

SetPosition
  TScrollBar-Methode 156
SetRange
  TScrollBar-Methode 155
  TScroller-Methode 122
SetSelIndex
  TListBox-Methode 147
SetSelString
  TListBox-Methode 147
SetTExt
  TStatic-Methode 149
SetupWindow
  Steuerelementobjekte und 140
  TMDIWindow-Methode 178
  TWindow-Methode 65
  wm_Create-Botschaft 96
Show
  TWindowsObject-Methode 97
ShowList
  TComboBox-Methode 168
Sicherheitsbereich
  Größe 267
Speicher
  globaler 199
  Großverbraucher 269
  Handles 10
  lokal vs. global 185
  lokaler 199
  sichere Zuweisung 267
  Sicherheitsbereich 267
  Sperre aufheben 185
  sperren 185
  Verwaltung 10
  verwerfbarer 184
Speicherknappheit 194
Speicherverwaltung
  Handles und 185
  Kollektionen und 282
  Multitasking 183
Speicherverwaltung 183ff.
Sperrenzähler 187, 192
  überprüfen 189
ss_Left-Stil 148
ss_NoPrefix-Stil 148
ss_Simple-Stil 149
Standard
  -Botschaftsantworten 84
  -Dialog 76
  -Fenster 77
  -Botschaftsantwort-Methoden 104
Statische Steuerelemente 147
  ändern 149
  Informationen über 148
  löschen 149
  Stile 147
  Transferpuffer 170
  unterstreichen 148
Status
  Fehlermeldung 269
  TStream-Feld 289
  TWindowsObject-Methode 268
StdDlgs-Unit 36, 77
  Ressourcen 134f.
STDDLGS.RES-Datei 134f.
StdWnds-Unit 115
Steuerelemente 137
  als abhängige untergeordnete Fenster 63
  als Fenster 62
  als untergeordnete Fenster 128
  anzeigen 140
  Benachrichtigungs-botschaften 129, 141
  Botschaften und 84f., 128, 140
  Botschaftsbeantwortung 65
  Dialogfenster und 128
  Ereignisse und 65
  erstellen 139
  Fenster mit 59
  Fenster und 138
  freigeben 140
  Handles 129
  IDs 64
  initialisieren 171
  manipulieren 141
  Ressourcen und 63
  Standard-Botschafts-verarbeitung 131
  Verwaltung 64
  Werte setzen 168
  Zurodnung von Objekten 131
Steuerelementobjekte 74, 138
  als Felder 63
  Anzeige 65
  Definition 131, 138
  Dialogfenster und 129
  konstruieren 64, 131, 139, 170
  Manipulation 128
  Stile 140
Stifte
  Bildschirmkontext und 219
  erzeugen 38
  Linienstile 223
  virtuelle 223

Stile 78ff.
  Aktionsschalter 149
  Bildlaufleisten 154
  Editierfelder 159
  Fenster- 108ff.
  Klassen 114
  Kombinationsfenster 167
  kombinieren 79
  Linien 223
  Steuerelementobjekte 140
  virtuelle Pinsel 224
Store
  TStreamRec-Feld 291
Store-Methode 289, 292
  hinzufügen 294
Streams 285ff.
  Destruktor 289
  Direktzugriff 287, 300
  DOS 287
  EMS 288
  entwerfen 300
  Fehler 301
  Fehlerbehandlung 289
  gepufferte 288
  indizierte 288
  kopieren 299
  lesen aus 288
  Load-Methode und 289
  nebengeordnete Fenster und 298
  Nicht-Objekte und 300
  nil-Objekte und 293
  Objekt-ID 291
  Objekte und 285ff.
  polymorphe 286f.
  Registrierung 286, 290, 292
  schreiben in 288, 296
  Store-Methode und 289
  Typenprüfung 294
  überlassen 300
  untergeordnete Fenster
  und 289, 297
  virtuelle Methodentabellen
  und 286
  vs. Dateien 285
Stringkollektionen 278
Stringtabellen
  Dialogfenster und 255
  IDs 255
  laden 255
  Menüs und 256
Style
  TWindowAttr-Feld 110
Symbole
  Fenster 112
  Mauszeiger vs. 254
  vordefinierte 114, 255

T
TApplication
  Objekte 73
TApplication-Objekt 87
TBufStream-Objekt 288
TButton-Objekt 149
TCheckBox-Objekt 150
TCollection-Objekt 271
TComboBox-Objekt 165
TControl-Objekt 138
TControl-Typ 62
TDosStream-Objekt 288
TEditWindow-Objekt 115
TEmsStream-Objekt 288
Text
  editieren 115
  zeichnen 30
TextOut-Funktion 31, 233
tf_GetData-Konstante 172
tf_SetData-Konstante 172
TFileDialog-Objekt 135
TFileWindow-Objekt 115
TileChildren
  TMDIWindow-Methode 179
TInputDialog-Objekt 134
Titel
  Hauptfenster 89, 91
TListBox-Objekt 143
TLogBrush-Record 222
TLogFont-Record 222
TLogPen-Record 222
TMDIClient-Objekt 176
TMDIWindow-Objekt 176
TMessage-Record 101
TMessage-Typ 25, 31
Toggle
  TCheckBox-Methode 152
TRadioButton-Objekt 150
Transfer
  Rückgabewert 172
  TControl-Methode 172
  TWindowsObject-Methode 172
TransferData
  TWindowsObject-Methode 172
Transferpuffer
  aktualisieren 171
  Auswahlfelder 170

Beispiel 172
Bildlaufleisten 170
definieren 169
Editierfelder 170
Kombinationsfenster 169
Listen 170
statische Steuerelemente 170
Truncate
TStream-Methode 300
TScrollBar-Objekt 154
TScroller
Attribute 118
Bereich 118
Einheiten 118
Seitengröße 118
Zeilengröße 118
TScroller-Objekt 118
TSortedCollection-Objekt 277
TStrCollection-Objekt 273
TStream-Objekt 288
TWindow 21, 23
Objekte 73
TWindow-Objekt 107
TWindowAttr-Record 108
TWindowsObject 23, 73
TWindowsObject-Typ 95
TWndClass
Standardwerte 113
TWndClass-Record 112f.
Typ-Umwandlung 31
Typen
Windows 76
Windows-Daten 13
Typenprüfung
Dateien und 286
Kollektionen und 272

U

Übergeordnete Fenster 24, 49, 97
Streams und 297
Vorfahre vs. 50
Uncheck
TCheckBox-Methode 152
Units
DLLs und 199
ObjectWindows und 59, 76
StdDlgs 77
WinProcs 76
WinTypes 76
WObjects 21, 76
Untergeordnete Fenster 24, 97
als Objektfelder 63
Attribute setzen 110
Destruktor 98
Dialogfenster als 125
ID-Bereich 103
Konstruktion 97
Liste mit 97, 139
MDI 175ff.
selbständige 50
Streams und 288, 297
unselbständige 50
Untergordnete Fenster 49
USER.EXE 16

V

Vererbung
Streams und 289
Virtuelle Methodentabellen
Streams und 291
virtuelle Pinsel 224
virtuelle Schriften 225
VmtLink
TStreamRec-Feld 291

W

Wahlfelder
hinzufügen 153
Informationen über 151
konstruieren 151
Status ändern 152
Windows
API 77
API-Arufruf 14
API-Konstanten 78
Datenstrukturen 78
Funktionen 77ff.
Koordinatensystem 30
Paletten-Manager 239
Struktur 15
Zwischenablage 203
WinProcs-Unit 76
WinTypes-Unit 76
wm_Close-Botschaft 93
wm_Command-Botschaft 102
wm_Compacting-Botschaft 195
wm_Create-Botschaft 96
wm_DDE_Advise-Botschaft 211
wm_DDE_Data-Botschaft 211
wm_DDE_Execute-Botschaft 212
wm_DDE_Initiate-Botschaft 209f.
wm_DDE_Poke-Botschaft 212, 216
wm_DDE_Request-Botschaft 211
wm_DDE_Terminate-Botschaft 209ff.

wm_DDE_Unadvise-Botschaft 212
wm_DestroyClipboard-Botschaft 207
wm_First-Konstante
  cm_First vs. 45
wm_HScroll-Botschaft 156
wm_LButtonDown-Botschaft 25
wm_LButtonUp-Botschaft 33
wm_MDIActivate-Botschaft 179
wm_MouseMove-Botschaft 33
wm_RButtonDown-Botschaft 25
wm_RenderAllFormats-
  Botschaft 207
wm_SetFocus-Methode 116
wm_VScroll-Botschaft 156
WMSetFocus
  TEditWindow-Methode 116
WMSize
  TEditWindow-Methode 116
WObject-Unit 76
wParam
  Botschaftsparameter 101
Write
  TStream-Methode 300
  TStream-Prozedur 292
ws_Child-Stil 143, 148, 154, 167
ws_HScroll-Konstante 119
ws_HScroll-Stil 159
ws_Visible-Stil 97, 143, 148,
  154, 167
ws_VScroll-Konstante 119
ws_VScroll-Stil 159

X
XUnit
  TScroller-Feld 118

Y
YUnit
  TScroller-Feld 118

Z
Zeichenfunktionen (GDI) 233
Zeichenwerkzeuge 220ff.
  Attribute 222
  erzeugen 37
  Handles 36
  Standardsatz 36
  verwenden 231
  virtuelle 223
  vordefinierte 222
  zeichnen und 40
Zeichnen
  geometrische Formen 236
  Text 30
  Zeichenwerkzeuge und 40
Zeiger 112
Zuordnung
  Botschaften und 131
Zwischenablage 12, 203
  Daten laden aus 205
  Editierfelder und 160
  Formate 203
  Speichern von Daten in 204

WINDOWS-
PROGRAMMIER-
HANDBUCH

TURBO PASCAL®

für Windows

BORLAND

BORLAND GMBH, LINDWURMSTRASSE 88, POSTFACH 15 03 40, D-8000 MÜNCHEN 2, TELEFON 0 89/7 20 10-0, TELEFAX 0 89/77 93 39, TELEX 5 212 637 mcm d,
CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001. (408) 438-530. OFFICES IN:
AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINDOM * ARTIKEL-NR. 112303